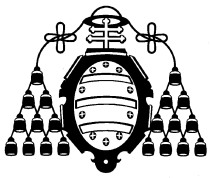
UNIVERSIDAD DE OVIEDO



DEPARTAMENTO DE INFORMÁTICA

Soluciones Metaheurísticas al "Job-Shop Scheduling Problem with Sequence-Dependent Setup Times"

Tesis Doctoral

Autor: Miguel Ángel González Fernández

Directora: María del Camino Rodríguez Vela

Julio, 2011

Agradecimientos

- A mi familia y amigos, por estar siempre ahí, especialmente a mis padres y a mi hermano.
- A los miembros del Grupo de Tecnologías de la Computación por su ayuda y apoyo.
- Y muy especialmente a mi tutora, María del Camino Rodríguez Vela, ya que sin ella hubiera sido imposible el desarrollo de esta tesis. No sólo por su ánimo y paciencia conmigo, sino también por la inestimable ayuda que me prestó y que me sigue prestando, por su amistad, y por todo el tiempo que me ha dedicado.
- Este trabajo ha sido financiado por el Ministerio Español de Educación y Ciencia mediante los proyectos de investigación MEC-FEDER TIN2007-67466-C02-01 y TIN2010-20976-C02-02, y por FICYT mediante la ayuda BP07-109. Agradezco también a la Unidad de Consultoría Estadística de la Universidad de Oviedo parte del análisis estadístico detallado realizado en el capítulo 9.

Resumen

Los problemas de scheduling requieren organizar en el tiempo la ejecución de tareas que comparten un conjunto finito de recursos, y que están sujetas a un conjunto de restricciones impuestas por diversos factores, como por ejemplo las características físicas del entorno, relaciones temporales o la normativa laboral. Este tipo de problemas aparecen con frecuencia en la vida real en numerosos entornos productivos y de servicios. El problema consiste en optimizar uno o varios criterios que se representan mediante funciones objetivo y que suelen estar relacionados con el coste o el tiempo total de ejecución. Algunos ejemplos de problemas de scheduling son los siguientes:

- Fabricación de obleas para circuitos semiconductores, donde cada oblea precisa de una serie de tareas como limpieza, oxidación, metalización, etc. Los objetivos pueden ser maximizar la utilización de algunas máquinas que son cuello de botella o minimizar el tiempo de ejecución.

- Planificar el aterrizaje de un conjunto de aviones sujetos a restricciones temporales que dependen de las características de los aviones. Los objetivos pueden ser minimizar la penalización por desvío con respecto al horario planeado para los aviones o maximizar las condiciones de seguridad.

- Enrutamiento de paquetes de datos a través de líneas de comunicación, donde se trata de maximizar el uso de la red y de minimizar los tiempos de llegada de los mensajes.

Todos estos problemas son de naturaleza combinatoria, es decir que hay que elegir una entre un conjunto exponencialmente grande de combinaciones posibles, y por lo tanto los problemas de scheduling precisan de algoritmos de búsqueda inteligentes para encontrar soluciones aceptables en un tiempo razonable. En la literatura se pueden encontrar aproximaciones a los problemas de scheduling basadas en todas las metaheurísticas conocidas.

En esta tesis nos centramos en el problema Job Shop Scheduling with Sequence Dependent Setup Times. Este problema es una generalización del problema Job Shop clásico que

tiene más interés en muchas aplicaciones reales. Además, es un problema mucho menos estudiado y su resolución es más compleja, ya que los tiempos de setup cambian la naturaleza del problema y muchas de las propiedades formales demostradas para el Job Shop, dejan de cumplirse en presencia de tiempos de setup. Por este motivo, la adaptación de las técnicas utilizadas para resolver el Job Shop muchas veces no será trivial.

Como técnicas de búsqueda utilizaremos los algoritmos genéticos y la búsqueda local. Es bien sabido que estos métodos no necesariamente producen la solución óptima del problema, pero hay que tener en cuenta que cuando el espacio de búsqueda del problema es enorme, como ocurre en este caso, los métodos exactos de resolución suelen ser demasiado costosos, ya sea en tiempo de ejecución o en recursos computacionales. El objetivo, por lo tanto, será diseñar estrategias eficaces para obtener buenas soluciones en un tiempo razonable. Para ello se deberán estudiar aspectos como heurísticos, estructuras de vecindad, propiedades del camino crítico, algoritmos de estimación de vecinos, grafos disyuntivos, constructores de planificaciones, y en general encontrar la mejor configuración para los métodos utilizados.

En esta tesis estudiaremos diversas funciones objetivo. La función objetivo a la que los investigadores han prestado mayor atención es sin duda el makespan, o tiempo de finalización de la última tarea. Pero otras funciones objetivo pueden tener mayor interés en problemas reales, en donde es posible que cada uno de los trabajos tenga un tiempo deseable de fin diferente, e incluso que algunos trabajos tengan una mayor prioridad que otros. Por este motivo, además de la minimización del makespan trataremos la minimización de otras tres funciones objetivo: maximum lateness, weighted tardiness y total flow time.

Las aportaciones principales de esta tesis serán el estudio de las diferencias entre las propiedades del job shop clásico y el job shop con tiempos de setup, la definición de un modelo de grafo disyuntivo para cada función objetivo estudiada, el diseño de una serie de estructuras de vecindad con sus respectivas condiciones de factibilidad y de no mejora, algoritmos de estimación de vecinos para diferentes funciones objetivo y otras consideraciones sobre cómo realizar una búsqueda local de la forma lo más eficiente posible, y por último, unos resultados experimentales que mostrarán que la correcta combinación de los métodos propuestos es capaz de mejorar en muchos casos a los mejores resultados conocidos en la literatura.

Índice de Contenidos

Lista de Figuras

Índice de Tablas

Índice de Algoritmos

Capítulo 1

INTRODUCCIÓN

1.1. Introducción al JSP y al SDST-JSP

El Job Shop Scheduling Problem (JSP) es un paradigma de la familia de problemas de optimización combinatoria y de satisfacción de restricciones. Esta familia de problemas ha interesado a los investigadores durante las últimas décadas debido a su complejidad y a la variedad de situaciones reales que modelan. Los problemas de scheduling implican, en general, asignar una serie de recursos a una serie de tareas, satisfaciendo unas restricciones y uno o varios criterios de optimización [29, 115]. Normalmente dichos recursos son limitados, y por lo tanto las tareas se asignan a los recursos en orden temporal. Desde un punto de vista económico, los recursos limitados se supone que escasean, y por lo tanto el problema de la planificación de tareas tiene más importancia que la meramente académica. El JSP es un problema clásico, que debido a su presencia en el mundo real, principalmente en los sectores industrial y de servicios, ha despertado mucho interés entre los investigadores. En la sección 1.2 describiremos varios ejemplos de problemas reales de scheduling.

Los recursos y las tareas pueden tomar diferentes formas. Así pues, por ejemplo, los recursos pueden ser máquinas en un taller, pistas de aterrizaje en un aeropuerto, el equipo de trabajo en una obra en construcción, las unidades de procesamiento en un entorno de computación, y así sucesivamente. Las tareas pueden ser unidades de trabajo en un proceso de producción, los despegues y aterrizajes en un aeropuerto, las fases de un proyecto de construcción, ejecución de programas de ordenador, etc. Cada tarea puede tener asignada

una prioridad, y un tiempo de inicio más temprano y un tiempo de fin límite. Los objetivos también pueden tomar diversas formas, por ejemplo, la minimización del tiempo de fin de la última tarea (éste es el criterio de optimización más típico y se le conoce con el nombre de makespan), o la minimización del tiempo que las tareas terminan tras su tiempo de fin límite (criterio conocido como minimización del tardiness, o del weighted tardiness si asignamos una determinada prioridad a cada tarea). Otras posibles funciones objetivo pueden ser el maximum lateness (en donde intentaremos minimizar el máximo incumplimiento de tiempo de fin límite) o el total flow time (que consistirá en minimizar la suma de los tiempos de finalización de todos los trabajos).

Debido a la complejidad de estos problemas, el desarrollo de métodos exactos que encuentran la solución óptima en un tiempo razonable es muy difícil. Por lo tanto, el esfuerzo se suele centrar en métodos heurísticos que encuentren de manera eficiente soluciones lo más cercanas posibles al óptimo.

En las últimas décadas, los problemas de planificación han sido investigados de forma intensiva debido a sus múltiples aplicaciones en areas de la industria, economía o ciencia, tal y como comentan Brucker y Knust en [32]. El job shop es NP-completo (demostrado por Garey et al. en [54]), y en general se considera uno de los problemas de optimización combinatoria más difíciles. Hasta el momento no hay métodos para resolver de forma óptima instancias de tamaño medio o grande.

En un problema job shop, el número total de planificaciones posibles es $(N!)^M$, siendo N el número de trabajos, y M el número de recursos, y por este motivo es imposible una exploración exhaustiva del espacio de búsqueda en un tiempo razonable, a no ser que el problema sea extremadamente pequeño. Por ejemplo, para un problema con 5 trabajos y 5 recursos el número total de planificaciones posibles ya sería de casi 25000 millones.

El Job Shop Scheduling Problem with Sequence-Dependent Setup Times (SDST-JSP) es una generalización del clásico Job Shop Scheduling Problem (JSP) en el cual se requiere una operación de mantenimiento o setup al pasar de un trabajo a otro en la misma máquina. De esta forma, el SDST-JSP modela mejor que el JSP muchas situaciones reales, como ocurre por ejemplo en algunas situaciones de la industria de semiconductores, tal y como comentan Balas et al. en [20].

La consideración de los tiempos de setup es una característica relevante en muchos problemas de scheduling reales y contribuye a aumentar la dificultad de resolver dichos problemas, respecto a sus variantes sin tiempos de setup. Por estas razones, se han realizado muchos

intentos de modelar y resolver problemas de scheduling con tiempos de setup. En general, una operación de setup en una planta de producción consiste en una operación que deja a una máquina lista para realizar otra actividad, después de que otra actividad haya sido completada en la misma máquina. En el caso más complejo, el tiempo de setup requerido depende no sólo de la máquina, sino también de la operación que ha terminado y de la que va a comenzar. En este caso hablaríamos de sequence-dependent setup times. Los problemas de scheduling de este tipo surgen, por ejemplo, en las industrias de automoviles, impresión, semiconductores, químicas o farmacéuticas, como señalan Ovacik y Uzsoy en [110] o Allahverdi et al. en [6]. También tenemos un posible ejemplo en operaciones de pintura, en donde diferentes colores pueden requerir distintos niveles de limpieza según el color de la siguiente operación de pintado, y por lo general es conveniente ir desde los colores claros hacia los oscuros.

Incorporar sequence-dependent setup times cambia de manera significativa la naturaleza de un problema de scheduling. Esto hace que las técnicas y resultados desarrollados para el JSP no sean aplicables directamente al SDST-JSP, por lo que las propiedades formales y los métodos de resolución deben ser reconsiderados, tal y como indican Zoghby et al. en [166] y Balas et al. en [20]. Sin embargo, ésto no es óbice para que los métodos ampliamente probados y que han mostrado su eficacia en el JSP clásico sean la base sobre la que se diseñan estrategias de resolución para tratar con tiempos de setup, como hacen por ejemplo Brucker y Thiele en [33], Artigues y Feillet en [15], o Vela et al. en [151], para la minimización del makespan en el SDST-JSP.

El hecho de considerar tiempos de setup en problemas job shop comenzó en 1969, cuando Wilbrecht y Prescott descubrieron a través de simulaciones que los tiempos de setup dependientes de la secuencia juegan un papel crítico en las instancias de job shop que operan cerca de su capacidad máxima (ver [157]). A pesar de este descubrimiento, como indican Allahverdi et al. en [5], en las siguientes tres décadas la mayoría de la investigación en scheduling asume los tiempos de setup como nulos o como parte del tiempo de ejecución de las tareas. Esta suposición simplifica el análisis y es capaz de modelar algunas aplicaciones, pero es negativa para la calidad de las soluciones en muchas otras aplicaciones que sí requieren el tratamiento explícito de los tiempos de setup. Estos autores proponen clasificar los problemas de scheduling con tiempos de setup basándose en características como batch o no-batch, y tiempos de setup dependientes o independientes de la secuencia. También mencionan algunas de las variantes más relevantes, como single machine, parallel machines, flow shops

y job shops, con funciones objetivo como el makespan, maximum lateness, tardiness, total setup, changeover cost, etc. Más recientemente, Allahverdi et al. en [6] actualizan el estudio de la investigación sobre tiempos de setup con los trabajos aparecidos a partir del año 2000, ya que los trabajos anteriores a esa fecha ya se revisaban en [5].

1.2. Ejemplos reales de problemas de scheduling

Ingolotti en [71], Pinedo en [115] o Sierra en [125] describen varios ejemplos de problemas de esta naturaleza. A continuación explicamos brevemente algunos de los más interesantes.

Fabrica de Bolsas de Papel. Consideremos una fábrica dedicada a producir bolsas de papel. El material básico para cada una de las operaciones son rollos de papel. El proceso productivo consiste en tres fases: impresión del logo, pegado de los laterales de la bolsa, y la costura del final o de ambos finales de la bolsa. Cada fase consiste en una serie de máquinas no necesariamente idénticas. Las máquinas de una fase se pueden diferenciar en la velocidad a la que operan, el número de colores que pueden imprimir o el tamaño de la bolsa que producen. Cada orden de producción indica la cantidad de bolsas de cada tipo a fabricar y a entregar en una fecha determinada. Los tiempos de procesamiento de las diferentes operaciones son proporcionales al tamaño de la orden, es decir al número de bolsas en la orden.

Un retraso en la entrega implica una sanción en forma de pérdida de confianza. La magnitud de la penalización depende de la importancia de la orden o del cliente y del retraso en la entrega. Uno de los objetivos del sistema de planificación es minimizar la suma de estas sanciones.

Cuando una máquina cambia de un tipo de bolsa a otro se requiere un tiempo de puesta en marcha o de configuración o setup. La longitud del tiempo de setup depende de las similitudes entre dos órdenes consecutivas, por ejemplo el número de colores en común, las diferencias en tamaños, etc. Un objetivo importante del sistema de planificación es la minimización del tiempo total de setups.

Fabricación de Semiconductores. Los semiconductores se fabrican en instalaciones altamente especializadas. Éste es el caso de los chips de memoria y los microprocesadores. El proceso de producción en estas instalaciones generalmente consta de cuatro fases: fabricación de la oblea, sonda de la oblea, el montaje o embalaje, y la prueba final. La fabricación de las obleas es la fase más compleja. Para producir los circuitos, las capas de metal y el material de

la oblea se construyen con patrones sobre las obleas de silicio o arseniuro de galio. Cada capa requiere una serie de operaciones que normalmente incluyen: limpieza, oxidación, deposición y metalización, la litografía, el grabado, la implantación de iones, extracción fotorresistente, y la inspección y medición. Como consiste en varias capas, cada oblea debe someterse a estas operaciones varias veces, por tanto hay una cantidad de recirculación en el proceso muy elevada. Las obleas se mueven a través de la instalación en lotes de un determinado tamaño. Algunos equipos pueden requerir de un tiempo de puesta en marcha o de setup entre lotes. Este tiempo depende de la configuración del lote que acaba de terminar y de la configuración del lote a punto de comenzar.

El número de pedidos en el proceso de producción es a menudo de cientos, teniendo cada uno su propia fecha de inicio y de envío o entrega. Uno de los objetivos de la planificación es cumplir las fechas de envío, en la medida en que sea posible, maximizando el rendimiento. El principal objetivo es maximizar la utilización del equipo, especialmente de las máquinas que son cuellos de botella, logrando también minimizar los tiempos de espera y de setup.

Asignación de Puertas de Embarque en un Aeropuerto. Las terminales de los aeropuertos importantes tienen multitud de puertas de embarque, y cientos de aviones que despegan y aterrizan cada día. Las puertas de embarque no son todas iguales, y tampoco lo son los aviones. Algunas puertas se encuentran en lugares espaciosos en los que los aviones grandes pueden acomodarse con facilidad. Sin embargo, otras puertas se encuentran en lugares de difícil acceso para los aviones, teniendo incluso que remolcar a muchos de ellos a sus puertas. Los aviones despegan y aterrizan con un calendario determinado. Sin embargo, dicha programación esta sujeta a ciertas variaciones aleatorias debidas al clima, a acontecimientos imprevistos, etc. Durante el tiempo que un avión permanece en una puerta de embarque, los pasajeros que llegan se han de bajar del avión, el avión ha de ser preparado y los pasajeros que salen han de embarcar. La hora prevista de salida puede verse como una fecha de vencimiento, siendo este parámetro lo que mide el rendimiento de la línea aérea. Sin embargo, si se sabe de antemano que el avión no puede aterrizar en su próximo aeropuerto de destino, debido a la congestión prevista en dicho aeropuerto a la hora de su aterrizaje, el avión no despega por política de ahorro de combustible. Si un avión no puede despegar, la política de funcionamiento es que los pasajeros permanezcan en la terminal y no en el avión. El aplazamiento de un embarque hace que el avión permanezca en la puerta un tiempo prolongado, por lo que otros aviones no pueden utilizarla. La planificación ha de asignar los aviones a las puertas de embarque de modo que dicha asignación sea físicamente posible (las

puertas estén disponibles en los tiempos de aterrizaje previstos) y se optimicen una serie de objetivos.

El objetivo de la planificación es minimizar tanto el trabajo del personal de la aerolínea como los retrasos del avión. En este entorno las puertas son los recursos y el manejo y prestación de servicios de los aviones las tareas. La llegada de un avión a una puerta representa el tiempo de inicio de una tarea, y la salida su tiempo de fin.

Planificación de Tareas en una *CPU*. Una de las funciones de un sistema operativo multitarea en un ordenador es decidir el tiempo que la *CPU* dedica a los diferentes programas que han de ser ejecutados. Generalmente, no se conoce de antemano el tiempo de ejecución exacto de cada uno de ellos. Sin embargo, una distribución aproximada de dichos tiempos de ejecución sí puede ser conocida de antemano, teniendo en cuenta sus medias y variaciones. Además, cada tarea por lo general tiene un nivel de prioridad determinado, aunque el sistema operativo permite a los usuarios establecer la prioridad de cada tarea. En este caso, el objetivo es minimizar la suma de los tiempos ponderados de fin de todas las tareas. Para evitar que una tarea breve permanezca mucho tiempo divide cada tarea en pequeños trozos. Una vez hecho esto, el sistema operativo rota los trozos en la *CPU*, de modo que en cualquier intervalo de tiempo la *CPU* pase cierta cantidad de tiempo con cada tarea. De esta manera, si el tiempo de procesamiento de una de las tareas es muy corto, la tarea será capaz de dejar el sistema rápidamente. La interrupción de una tarea en ejecución se conoce como expulsión. Es evidente que una política óptima en este entorno hace uso intensivo de expulsiones. En la práctica sucede con frecuencia que la elección de una planificación tiene un impacto significativo en el rendimiento del sistema, por lo que tiene sentido dedicar tiempo y esfuerzo a buscar una buena planificación, en lugar de elegir simplemente una al azar.

Planificación de Horarios Ferroviarios. El problema de Optimizar y Programar horarios para Trenes (*OPT*) es un problema real que puede ser considerado como un problema de scheduling similar al job shop. El sistema de tráfico ferroviario es muy complejo, por lo que el proceso de planificación se suele dividir en varios pasos, siendo la planificación de horarios para trenes uno de estos pasos. La formalización y resolución de este problema como un *JSP* puede verse con mayor nivel de detalle en la tesis doctoral de Ingolotti [71].

En el problema *OPT* se considera un conjunto de trenes y una línea ferroviaria, la cual estará formada por una secuencia ordenada de dependencias (apeaderos, estaciones, bifurcaciones, etc.). Cada tren tiene definido un recorrido sobre la misma, que no es el mismo para todos los trenes. El orden entre las dependencias determina el orden en el cual

deben ser visitadas por los trenes que viajan en un determinado sentido, que se denomina *ida*. Cuando los trenes viajan en sentido contrario se denomina *vuelta*.

El problema consiste en asignar un horario factible a cada tren, cumpliendo una serie de restricciones y optimizando una determinada función objetivo. Las restricciones que debe cumplir la planificación solución tienen en cuenta la capacidad de la infraestructura ferroviaria (señalización de cada tramo y número de vias en la dependencia y entre dependencias consecutivas), el tráfico de la línea (gestión de varios trenes por línea y prevención de colisiones), su mantenimiento (horario de cierre de estaciones, tiempos de mantenimiento), el servicio al cliente (frecuencia de salidas entre trenes, paradas en apeaderos) y las condiciones impuestas a la planificación (intervalos de salida y llegada de los trenes, y máximo retraso permitido sobre un horario para el tren).

Veamos ahora la correspondencia entre la definición del problema y su formulación como un *JSP*. Un *trabajo* es el recorrido completo que lleva a cabo un tren. Las *tareas* del trabajo son el paso de un tren por las estaciones o por los tramos entre estaciones. Para realizar cada tarea es necesario una serie de *recursos*, en este problema los recursos son las vías en la estación o las vías en el tramo, dependiendo del tipo de tarea. El *orden* de las tareas en un trabajo viene determinado por el recorrido de cada tren. La *duración* de una tarea viene determinada por el tiempo que permanece un tren en una estación o por el tiempo que un tren emplea en recorrer un tramo de su recorrido. Los trenes de *ida* emplean los recursos en un determinado orden, los de *vuelta* emplean los mismos recursos, o una parte de ellos, en el orden inverso.

Las restricciones del problema determinan las condiciones en que es factible la utilización de los recursos, teniendo en cuenta todas las tareas del problema. Por ejemplo, las restricciones de cruce regulan el uso de recursos utilizados por tareas pertenecientes a trabajos que siguen un orden inverso; así pues estas tareas no podrán utilizar a la vez el mismo recurso debiendo existir un margen de seguridad (tiempo de setup) entre operaciones que emplean el mismo recurso, así como entre el intercambio de recursos entre operaciones. Otras restricciones regulan la utilización de recursos por operaciones que siguen el mismo orden, algunos permiten ser utilizados por más de una tarea siempre que exista un intervalo de tiempo entre sus tiempos de inicio y fin, mientras que otros no permiten que dos tareas lo utilicen simultáneamente. Las estaciones son centros de trabajo que no pueden llevar a cabo más tareas que el número de máquinas (vías) disponibles. El período de tiempo que una tarea puede utilizar un recurso, o que un recurso puede ser utilizado teniendo en cuenta su

capacidad total, está también limitado por las operaciones de mantenimiento o el horario de cierre de una estación.

El objetivo de este problema es minimizar el retraso promedio de cada trabajo con respecto a un tiempo de referencia (horario) y la diferencia entre el retraso promedio correspondiente a los trabajos que siguen un orden de utilización de recursos y el retraso correspondiente a los trabajos que siguen el orden inverso. A cada uno de estos criterios se le asigna un peso.

1.3. Revisión bibliográfica

Como hemos indicado anteriormente, el JSP es un problema clásico que consiste en la asignación de recursos a unas tareas, cumpliendo una serie de restricciones y optimizando una serie de objetivos, y el añadido de los tiempos de setup complica bastante el problema. Desde principios del siglo XXI se han empezado a tratar mucho más los problemas de scheduling que consideran tiempos de setup, aunque en muchas ocasiones se tratan problemas diferentes al job shop.

Por ejemplo, para los problemas con una única máquina, Uzsoy et al. han aplicado diferentes estrategias a distintas funciones objetivo, entre otras programación dinámica incompleta, un algoritmo rolling horizon (que consiste en resolver una secuencia de subproblemas de forma óptima mediante un algoritmo de ramificación y poda, e implementar una parte de la solución obtenida), y un método de búsqueda local que comienza a partir de una solución obtenida mediante un heurístico clásico (ver [141]). También podemos citar los trabajos de Crauwels et al. en [43], Koulamas y Kyparisis en [77] o Baptiste y Le Pape en [22], que tratan la minimización de diversas funciones objetivo aplicando heurísticos de búsqueda local. También se ha tratado el single machine scheduling, por ejemplo, con el algoritmo genético híbrido propuesto por Miller et al. en [94] o con el algoritmo GRASP propuesto por Armentano y Basi de Araujo en [10]

Para el parallel machine scheduling se han propuesto varios métodos, por ejemplo basados en búsqueda tabú, como el de Bilge et al. en [28], GRASP con memoria adaptativa como el utilizado por Armentano y Filho en [11], enfriamiento simulado como el presentado por Lee y Pinedo en [83], o algoritmos genéticos como el de Fowler et al. del trabajo [53] o el de Vallada y Ruiz de [143]. Estas y otras aproximaciones son revisadas por Armentano y Filho en [11].

Otro problema de scheduling con tiempos de setup diferente al SDST-JSP que aparece en la literatura puede ser el heurístico basado en relajaciones del TSP propuesto por Rios-Mercado y Bard en [119] para el Flow Shop Scheduling Problem, en donde comparan sus resultados con un algoritmo GRASP. Otro ejemplo más reciente se puede encontrar en [100], en el que Naderi et al. tratan el problema Flexible Flow Lines con sequence-dependent setup times y con tiempos de mantenimiento para las máquinas.

En [34], Candido et al. consideran una extensión del JSP clásico con tiempos de setup y características adicionales de entornos reales, como planes de procesamiento alternativos, o recursos renovables. Proponen un algoritmo genético hibridizado con reglas de prioridad para la inicialización, y un método simple de escalada para la mejora de los cromosomas. La estructura de vecindad esta basada en cambios en el camino crítico y había sido previamente estudiada y formalizada para el JSP clásico. Obtienen vecinos de un modelo simplificado del problema, y después evalúan dichos vecinos en el problema real con setups.

En [166], Zoghby et al. proponen una búsqueda en vecindades con reparación heurística que extiende algunos métodos de búsqueda local para el JSP. Este método se aplica a instancias con características adicionales, como por ejemplo trabajos reentrantes.

A continuación revisaremos algunos de los avances en el SDST-JSP en cada una de las funciones objetivo que trataremos en esta tesis.

1.3.1. Minimización del makespan

El makespan es la función objetivo por excelencia, y en general es la más estudiada en la literatura. Los trabajos sobre la minimización del makespan en el SDST-JSP, en general, intentan extender métodos que habían dado buenos resultados en el JSP clásico. Por ejemplo, en [33], Brucker y Thiele proponen una regla de prioridad que es una extensión del algoritmo de Giffler y Thompson [55]. Utilizan esa regla dentro de su método de ramificación y poda, que es una extensión de otro algoritmo anterior para el JSP clásico, propuesto por Brucker et al. en [31]. También proponen un conjunto de instancias, denominadas conjunto BT, que desde entonces es un benchmark ampliamente utilizado. Los resultados obtenidos allí son mejorados por Artigues et al. en [17] con nuevas reglas de prioridad. Artigues y Feillet incorporan algunas de estas reglas a un algoritmo de ramificación y poda en [15]. Este algoritmo es capaz de resolver de forma óptima más instancias del conjunto BT que el método de Brucker y Thiele y también ha mejorado las mejores cotas inferiores y superiores en algunas de las instancias no resueltas de forma óptima. En [19], Balas et al. extienden el

heurístico shifting bottleneck propuesto por Adams et al. en [3], y las soluciones se mejoran mediante un procedimiento de búsqueda local guiada. El algoritmo genético propuesto por Cheung y Zhou en [40] también trata la minimización del makespan en el SDST-JSP y es una extensión de un algoritmo genético para el JSP.

1.3.2. Minimización del maximum lateness

En la sección anterior hemos hablado de minimizar el makespan, pero sin embargo es muy común que el objetivo principal de muchas operaciones de scheduling sea cumplir los tiempos de fin de cada trabajo, o due dates. Esto es importante por varios motivos: por una parte el proporcionar a tiempo un determinado producto o servicio nos lleva a tener contentos a más consumidores, y por otra parte el tener terminados a tiempo partes o productos semi-acabados para el siguiente proceso de manufactura nos lleva a tener una mejor producción y un mejor control y capacidad de planificación.

En [141] Uzsoy y Velásquez dejan clara la importancia que tiene el lateness como criterio de optimización. En concreto explican: "we take the position that the function of shop-floor scheduling is to ensure that production on the shop-floor adheres as closely as possible to the master production schedule, since the personnel making shop-floor decisions do not have information necessary to make high-level capacity allocation decisions such as which customers orders have priority over others. Hence we assume, following common industrial practice, that each job on the shop-floor has been assigned a completion date by the planning system, and that the scheduling system should try to minimize some function of deviation from these due dates. This motivates the formulation of the shop-floor scheduling problem as that of minimizing maximum lateness (L_{max}), since this will avoid making some jobs early at the expense of others being extremely delayed". En ese trabajo los autores establecen una relación entre el makespan y el maximum lateness, y de hecho su algoritmo de búsqueda local, denominado MEDD, trata de reducir el makespan esperando que dicha reducción también mejore el maximum lateness.

La minimización del maximum lateness en el SDST-JSP ha sido considerada por algunos investigadores. Por ejemplo, en [109] Ovacik y Uzsoy proponen una familia de heurísticos que utilizan información global para tomar decisiones locales a nivel de una máquina. Su estudio experimental demuestra que esos heurísticos son mucho más eficientes que un conjunto de reglas de prioridad miopes. En el mismo trabajo se propone un benchmark con algunas instancias inspiradas en el entorno de la industria de semiconductores. Los mismos autores

ofrecen más resultados en [110]. Dichos resultados son mejorados por Balas et al. en [20] con un algoritmo denominado SB-GLS que extiende el conocido heurístico shifting bottleneck propuesto por Adams et al. en [3] y lo combina con un procedimiento de búsqueda local guiada. Más recientemente, en [107] Oddi et al. proponen un método Iterative-Sampling Search (ISS), que es una búsqueda basada en restricciones aplicada dentro de un esquema de muestreo iterativo (Iterative Sampling).

1.3.3. Minimización del weighted tardiness

El weighted tardiness es otra función objetivo relacionada con los due dates. A pesar de que el servicio al cliente con respecto a cumplir due dates, tiene cada vez una mayor importancia, los trabajos de investigación que tratan sobre la minimización del weighted tardiness en problemas job-shop es más bien escasa. De hecho, el único trabajo que conocemos sobre la minimización del weighted tardiness en el SDST-JSP es el que presentan Sun y Noble en [139]. Proponen un algoritmo shifting bottleneck y en el estudio experimental se comparan contra varias reglas de prioridad en instancias generadas de forma aleatoria. Debido a esta ausencia de trabajos sobre la minimización del weighted tardiness en el SDST-JSP, revisaremos a continuación varios trabajos que minimizan esta función objetivo en otros problemas, particularmente el JSP clásico sin tiempos de setup.

Por ejemplo, en [152] Vepsalainen y Morton desarrollan dos nuevas reglas de prioridad y las comparan con varias reglas de prioridad clásicas. En [133] Singer y Pinedo proponen un método shifting bottleneck que resuelve problemas de una única máquina utilizando también reglas de prioridad. En [12] Armentano y Scrich proponen, para el caso de pesos unitarios, un método de búsqueda tabú que utiliza estrategias de diversificación e intensificación. En [79] Kreipl propone un heurístico de tipo large step random walk que itera entre pasos largos y cortos. Los cortos se basan en un método de escalada, mientras que los largos se basan en el algoritmo de Metropolis y permiten al algoritmo escapar de óptimos locales. Mattfeld y Bierwirth han propuesto en [89] un algoritmo genético que tiene en cuenta varios criterios de tardiness. En [131] Singer propone un método de descomposición en ventanas de tiempos y se tratan instancias de gran tamaño. Uno de los métodos más recientes para la minimización del weighted tardiness en el JSP es el propuesto por Essafi et al. en [49], que consiste en un híbrido de algoritmo genético y búsqueda local. Su búsqueda local se basa en inversiones de arcos en el camino crítico e itera entre fases de mejora y de perturbación. Las fases de mejora consisten en un método de escalada, mientras que las de perturbación

eligen aleatoriamente un cierto número de movimientos de la vecindad, para poder escapar del máximo local encontrado. Realizan un estudio experimental bastante extenso y obtienen resultados muy competitivos. Otro método bastante reciente es el propuesto por Zhou et al. en [164]. En dicho trabajo combinan un algoritmo genético con tres heurísticos diferentes. El algoritmo genético determina la primera operación de cada máquina, y el heurístico decide el orden del resto de operaciones. En [45] DeBontridder propone una búsqueda tabú para minimizar el weighted tardiness en una generalización del job shop que incluye release times, lags de tiempo positivos entre tareas, y un grafo de precedencias generalizado. Utilizan una vecindad basada en la inversión de arcos críticos que se encuentran en el extremo de un bloque crítico. En el estudio experimental utilizan instancias del job shop clásico y se comparan con el método shifting bottleneck propuesto por Singer y Pinedo en [133] y con el método propuesto por Kreipl en [79], obteniendo resultados muy competitivos. También podemos destacar la búsqueda local propuesta por Mati et al. en [86], que en realidad no esta dedicada en exclusiva a la minimización del weighted tardiness, sino que su método es capaz de minimizar cualquier función objetivo regular, pero en dicho trabajo ofrecen resultados muy competitivos sobre esta función objetivo. En [146], Van Hentenryck y Michel también presentan resultados de su método de búsqueda local sobre la minimización del weighted tardiness, con la intención de demostrar la eficacia de las abstracciones propuestas en dicho trabajo. Por otra parte, el único método exacto existente para el JSP es el de ramificación y poda propuesto por Singer y Pinedo en [132].

1.3.4. Minimización del total flow time

El total flow time es otra función objetivo que puede tener mucho interés en problemas reales, por ejemplo en aplicaciones en donde sea muy importante el servicio al cliente. Esta función no considera due dates, sino que consiste en minimizar la suma del tiempo de finalización de todos los trabajos. Uno de los problemas con el que nos encontramos aquí es que apenas se pueden encontrar en la literatura métodos específicos para resolverla, aunque hay que tener en cuenta que el total flow time es en realidad un caso particular de la función weighted tardiness, como explicaremos en otros capítulos. Como único ejemplo de publicación dedicada en exclusiva al total flow time podemos destacar los algoritmos de búsqueda heurística propuestos por Sierra et al. en [125, 126, 128, 129, 130], que tratan de minimizar el total flow time en el JSP clásico mediante un algoritmo A^*, y se describirán con detalle en la sección 4.4.2.

1.4. Motivaciones

La resolución del SDST-JSP supone un reto, ya que es un problema mucho menos estudiado que el JSP tradicional, y muchas de las propiedades y técnicas utilizadas para resolver el JSP no son válidas para resolver el SDST-JSP. Además el problema tiene una gran complejidad, ya que el job shop es NP-completo y en general se considera uno de los problemas de optimización combinatoria más difíciles. La inclusión de tiempos de setup lo hace un problema aún más difícil. El problema tiene gran interés porque se pueden modelar con él muchas situaciones reales. Por otra parte, el estudio de varias funciones objetivo para el problema es muy interesante, ya que diferentes aplicaciones prácticas pueden necesitar minimizar diferentes funciones objetivo, y es frecuente que se necesiten utilizar distintas técnicas de resolución para cada una de las funciones.

En esta tesis extenderemos diversos métodos de resolución que se habían aplicado previamente al JSP con bastante éxito, y trataremos de optimizar las siguientes funciones objetivo: makespan, maximum lateness, weighted tardiness y total flow time. Los métodos que utilizaremos para ello son los algoritmos genéticos y la búsqueda local. En [63], González et al. muestran resultados de un estudio experimental sobre un conjunto de instancias difíciles elegidas por Applegate y Cook en [9] y se puede ver que el algoritmo genético hibridizado con búsqueda local es competitivo con los métodos más eficientes para resolver el JSP.

En cuanto a la búsqueda local, trataremos desde las variantes simples como el máximo gradiente, hasta las más avanzadas como la búsqueda tabú, y también estudiaremos los efectos de su hibridación con un algoritmo genético. La búaqueda tabú ha obtenido resultados muy buenos en diversos problemas de scheduling, como se puede constatar por ejemplo en el algoritmo de Nowicki y Smutnicki en [106] o en el algoritmo de Zhang et al. en [162], que son dos de los mejores algoritmos conocidos para el *JSP*. En particular, para resolver el SDST-JSP deberemos adaptar heurísticos, estructuras de vecindad, propiedades del camino crítico, algoritmos de estimación de vecinos, grafos disyuntivos, constructores de planificaciones, y muchos otros aspectos. La extensión al SDST-JSP de todas estas cuestiones no será trivial debido a que los tiempos de setup modifican la naturaleza del problema.

Entonces, la motivación principal de esta tesis es lograr que la correcta combinación de los métodos propuestos sea capaz de superar los resultados obtenidos por los mejores métodos conocidos en la literatura.

1.5. Objetivos

Los principales objetivos de esta tesis son los siguientes:

1. Profundizar en las diferencias entre el JSP y el SDST-JSP, ya que la inclusión de los tiempos de setup hará que muchas propiedades sean diferentes entre esos dos problemas.

2. Construir los modelos de grafo disyuntivo para cada una de las cuatro funciones objetivo que se estudiaran: el makespan, el maximum lateness, el weighted tardiness y el total flow time.

3. Diseñar y comparar métodos de búsqueda local, algoritmos genéticos y constructores de planificaciones, con el objetivo de resolver el SDST-JSP de la forma más eficiente posible.

4. Adaptar estructuras de vecindad del JSP, con sus correspondientes condiciones de factibilidad, no mejora y algoritmos de estimación, para que puedan tratar el SDST-JSP, o incluso proponer estructuras de vecindad novedosas. Tendremos en cuenta las diferencias que pueda haber según la función objetivo que se intente minimizar.

5. Evaluar las técnicas propuestas, utilizando como repositorios de instancias para cada función objetivo aquellos que nos permitan comparar nuestras técnicas con los mejores algoritmos conocidos, siempre que sea posible.

1.6. Estructura y principales contribuciones

A continuación mostramos un breve esquema de la organización del resto de la tesis, remarcando las aportaciones más relevantes al estado del arte.

En el capítulo 2 formulamos el SDST-JSP y detallamos la notación utilizada a lo largo de la tesis. Además introduciremos el modelo del grafo disyuntivo. La definición de un modelo de grafo disyuntivo para cada función objetivo es una de las aportaciones de la tesis. La importancia de la definición del grafo disyuntivo se hace patente especialmente en la minimización del maximum lateness, donde observamos que tanto los resultados formales como los métodos utilizados para la minimización del makespan son fácilmente adaptables a la nueva función objetivo. El estudio de las propiedades de los caminos críticos que se realiza a continuación es otra de las aportaciones de la tesis.

En el capítulo 3 explicaremos los diferentes tipos de planificaciones existentes, y los constructores de planificaciones que utilizaremos a lo largo de la tesis. En concreto veremos que la construcción de planificaciones activas en el SDST-JSP es más compleja que en el JSP. Para construir planificaciones activas hemos considerado el Serial Schedule Generation Schema propuesto por Artigues et al. en [17], y la extensión al SDST-JSP del algoritmo $G\&T$ del JSP clásico, denominada $EG\&T$ por Artigues y Lopez en [16].

En el capítulo 4 revisaremos algunos de los métodos clásicos de resolución para resolver este tipo de problemas de optimización. En este capítulo no se revisarán los algoritmos genéticos ni la búsqueda local porque, al ser los métodos utilizados en esta tesis, les dedicaremos sus propios capítulos. En concreto en el capítulo 4 nos centraremos en las reglas de prioridad, el método shifting bottleneck y diversos métodos exactos. En particular la búsqueda heurística en espacios de estados y los métodos de programación de restricciones. Aunque en esta tesis no utilizamos directamente esas técnicas, en muchos de los estudios experimentales compararemos nuestros algoritmos con ese tipo de estrategias, y por este motivo es relevante describirlas con más detalle.

El capítulo 5 detalla el algoritmo genético utilizado para el SDST-JSP, además de hacer una breve revisión bibliográfica sobre la aplicación de este tipo de algoritmos a problemas de scheduling. En concreto explicamos la estructura general de un genético, representación de los individuos, generación de la población inicial, los operadores genéticos de selección, cruce, mutación y fitness, y la forma de combinar un algoritmo genético con un método de búsqueda local.

El capítulo 6 describirá los algoritmos de búsqueda local utilizados en esta tesis. Estos algoritmos están inspirados en los ya desarrollados para el JSP por varios investigadores, por ejemplo el de Dell' Amico y Trubian en [46], el de Nowicki y Smutnicki en [105], el de Mattfeld en [88] o el de Jain et al. en [73]. También en este caso realizaremos una revisión bibliográfica sobre estos métodos. Describiremos en detalle tanto los algoritmos de escalada simple como otros más complejos. Entre estos últimos, uno de los heurísticos más populares es la búsqueda tabú (TS), presentada con detalle en el trabajo de Glover en [57] o en el trabajo de Glover y Laguna en [59]. La búsqueda tabú se basa en el uso sistemático de memoria adaptativa y exploración guiada, que permite llevar la búsqueda hacia regiones no exploradas del espacio de búsqueda, pudiendo escapar de óptimos locales. A lo largo de esta tesis también veremos que una posibilidad muy interesante es combinar la búsqueda tabú con los algoritmos genéticos.

En el capítulo 7 se definen las estructuras de vecindad utilizadas en esta tesis para resolver el SDST-JSP. Para estas estructuras estudiaremos sus respectivas condiciones de factibilidad y de no mejora, algoritmos de estimación de vecinos para diferentes funciones objetivo, y otras consideraciones sobre cómo realizar una búsqueda local de la forma más eficiente posible. Consideramos primero la estructura de vecindad denominada N_1 en la literatura, y formalizamos una extensión para el SDST-JSP que denominamos N_1^S. También hemos desarrollado otra estructura de vecindad que denominamos N^S, y que se inspira en la estructura N_2 para el JSP. Además, proponemos métodos para estimar la calidad de los vecinos generados, y comprobaremos que los algoritmos de estimación para las funciones objetivo de tipo suma son menos precisos y consumirán más tiempo de ejecución que algoritmos similares para el makespan o el maximum lateness, debido a la diferencia existente en la dificultad de resolución del problema. El capítulo también demuestra varias propiedades formales del JSP clásico que dejan de cumplirse con el añadido de los tiempos de setup, y además estudia el número de caminos críticos que es aconsejable considerar para generar vecinos. Este capítulo, junto con los resultados experimentales, es probablemente la mayor aportación de esta tesis.

Los capítulos 8, 9, 10 y 11 muestran los resultados obtenidos en el estudio experimental para las diferentes funciones objetivo estudiadas. Veremos que estos resultados muchas veces serán mejores que los mejores resultados conocidos en la literatura, por lo cual serán también una aportación muy importante.

El capítulo 8 expone una serie de estudios experimentales sobre la minimización del makespan en el SDST-JSP. Un primer paso será comprobar cuáles son las configuraciones más eficientes para los métodos propuestos a lo largo de esta tesis. Realizaremos diversos experimentos para elegir heurísticos, estructuras de vecindad y otras consideraciones. Posteriormente, para compararnos con el resto de métodos existentes en la literatura, realizamos estudios experimentales sobre varios benchmarks típicos. Uno de los benchmarks para evaluar la minimización del makespan es el conjunto de 15 instancias propuesto por Brucker y Thiele en [33] (el conjunto BT). En los resultados obtenidos en este benchmark se verá que podemos superar a los mejores métodos existentes en la actualidad, como el algoritmo de ramificación y poda propuesto por Artigues y Feillet en [15] o el método shifting bottleneck propuesto por Balas et al. en [19]. Mostraremos además resultados sobre el benchmark para la minimización del makespan propuesto por Vela et al. en [151], que dispone de instancias más grandes que las del conjunto BT. Estas instancias han sido generadas a partir del con-

junto de problemas elegidos por Applegate y Cook en [9] como difíciles de resolver para el JSP. También proporcionaremos resultados sobre otros benchmarks, como por ejemplo el propuesto por Cheung y Zhou en [40], además de resultados en instancias del JSP clásico sin tiempos de setup.

En el capítulo 9 realizamos un estudio experimental sobre la minimización del maximum lateness en el SDST-JSP. Nos compararemos con los mejores métodos conocidos hasta el momento, que por lo que sabemos son el algoritmo Shifting Bottleneck con Guided Local Search (SB+GLS) propuesto por Balas et al. en [20] y el algoritmo basado en Iterative Sampling Search (ISS) propuesto por Oddi et al. en [107]. Para ello, hemos utilizado las mismas instancias de benchmark que ellos han utilizado, es decir los conjuntos $i305$, $i315$, $i325$, $i605$, $i615$ e $i625$, propuestos por Ovacik y Uzsoy en [109]. En los resultados del estudio experimental se podrá ver que nuestros resultados mejoran a los mejores conocidos hasta el momento en condiciones de ejecución similares. En este estudio experimental, debido a que los benchmarks utilizados se componen de un gran número de instancias, se ha realizado un análisis estadístico más detallado dependiendo de los parámetros utilizados para generar las instancias. De los resultados de este análisis se concluirá que el método propuesto es más estable que los demás respecto a cambios en los parámetros que definen las instancias de los benchmarks.

En el capítulo 10 mostramos un estudio experimental sobre la minimización del weighted tardiness, tanto en el JSP como en el SDST-JSP. En el JSP clásico sin tiempos de setup nos compararemos, principalmente, con el método de búsqueda local propuesto por Kreipl en [79], el algoritmo genético híbrido propuesto por Essafi et al. en [49], y el método de búsqueda local propuesto por Mati et al. en [86]. En cuanto al SDST-JSP, el único algoritmo que conocemos es el shifting bottleneck propuesto por Sun y Noble en [139], y en el estudio experimental utilizan instancias generadas de forma aleatoria. Hemos contactado con los autores y ya no disponen de las instancias utilizadas en su estudio, por lo que no es posible hacer una comparación fiable con ese algoritmo. Por estos motivos, hemos optado por modelar el problema mediante el ILOG CPLEX CP Optimizer y comparar con él los resultados de nuestro método.

En el capítulo 11 realizamos un estudio experimental sobre la minimización del total flow time en benchmarks del JSP clásico y del SDST-JSP, para comparar nuestros métodos con otros algoritmos del estado del arte. En el JSP nos compararemos con los algoritmos de búsqueda heurística propuestos por Sierra et al. en [125, 129, 130] y con el algoritmo de

búsqueda local propuesto por Kreipl en [79]. Este último método en realidad esta diseñado para minimizar el weighted tardiness, sin embargo podemos compararnos con él gracias a que el total flow time es en realidad un caso particular del weighted tardiness. Los resultados mostrarán que el algoritmo propuesto es competitivo con esos dos métodos. Por otra parte, no conocemos trabajos que traten de minimizar el total flow time en el SDST-JSP, por lo que de nuevo decidimos compararnos con los resultados obtenidos por el ILOG CPLEX CP Optimizer.

Finalmente, en el capítulo 12 resumimos las principales conclusiones, las líneas de trabajo futuro que esta tesis deja abiertas, y las publicaciones derivadas del trabajo realizado en ella.

Capítulo 2

FORMULACIÓN DEL PROBLEMA

2.1. Introducción

En este capítulo realizaremos una descripción general de los problemas de scheduling, definiremos el Job Shop Scheduling Problem with Sequence-Dependent Setup Times (SDST-JSP), e introduciremos la notación utilizada a lo largo del trabajo. Esta notación se basa en el modelo de grafo disyuntivo propuesto por Roy y Sussmann en [123] para el problema Job Shop Scheduling clásico. Extenderemos esa representación para tener en cuenta los tiempos de setup, de una forma parecida a la que se puede ver en el trabajo de Zoghby et al. en [166]. También definiremos diferentes funciones objetivo para el problema, y la adaptación del grafo disyuntivo realizada para poder tratar cada una de ellas. También definiremos y comentaremos algunas propiedades de dos de los conceptos más importantes que aparecerán en la sección 7: camino crítico y bloque crítico.

2.2. Formulación general de los problemas de scheduling

Los problemas de scheduling son una subclase de problemas CSP (satisfacción de restricciones) que aparecen con frecuencia en entornos reales. Podemos encontrar problemas de esta familia en muchas áreas de la industria, la administración y la ciencia; problemas

que van desde la organización de la producción hasta la planificación de multiprocesadores, industrias de semiconductores, asignación de puertas de embarque en un aeropuerto, o planificación de horarios ferroviarios. Los problemas de scheduling son en general NP-duros, lo que implica que tienen gran complejidad y no se pueden resolver de manera eficiente mediante estrategias exactas. Para su resolución se han aplicado prácticamente todas las técnicas de Inteligencia Artificial, con mayor o menor éxito.

En los problemas de scheduling se tiene un conjunto de N trabajos $\{J_1, \ldots, J_N\}$ que han de ser procesados sobre un conjunto de M recursos o máquinas físicas $\{R_1, \ldots, R_M\}$. Una planificación consiste en encontrar para cada trabajo un tiempo o un intervalo de tiempos en los que éste puede procesarse en una o varias máquinas. El objetivo es encontrar una planificación sujeta a una serie de restricciones y que optimice una o varias funciones objetivo. El problema de planificación general se puede describir del siguiente modo:

- **General Shop Scheduling Problem** (GSSP): se tiene un conjunto de N trabajos $\{J_1, \ldots, J_N\}$ y un conjunto R de M recursos físicos o máquinas $\{R_1, \ldots, R_M\}$. Cada trabajo J_i consta de un conjunto de tareas u operaciones $\{\theta_{i1}, \ldots, \theta_{iM}\}$. Cada una de las operaciones θ_{ij} tiene un tiempo de procesamiento $p_{\theta_{ij}}$ y debe ser procesada sobre una única máquina del conjunto R. Cada trabajo sólo puede ser procesado sobre una máquina, y cada máquina sólo puede procesar un trabajo en un instante. El objetivo es encontrar una planificación factible que optimice una función objetivo que depende del tiempo de fin de los trabajos.

Los problemas que derivan del GSSP son casos especiales de éste que añaden alguna restricción. Los problemas derivados más referenciados en la literatura, ordenados de más general a más particular, son los siguientes:

- **Open Shop Scheduling Problem** (OSSP): es un GSSP en el que cada trabajo J_i consiste en M operaciones $\{\theta_{i1}, \ldots, \theta_{iM}\}$ con tiempos de procesamiento $p_{\theta_{ij}}$. Cada una de las tareas de un trabajo se procesará sobre una máquina diferente, no existiendo relaciones de precedencia entre las operaciones. El problema consiste en encontrar órdenes de los trabajos (ordenaciones de las operaciones pertenecientes al mismo trabajo) y órdenes de las máquinas (ordenaciones de las operaciones que han de ser procesadas en la misma máquina) que optimicen una cierta función objetivo.

- **Job Shop Scheduling Problem** (JSSP): es un GSSP en el que cada trabajo J_i consiste en M operaciones $\{\theta_{i1}, \ldots, \theta_{iM}\}$ con tiempos de procesamiento $p_{\theta_{ij}}$ donde

cada tarea $\theta_{ij}(j = 1, \ldots, M)$ debe ser procesada en una máquina distinta. En este caso también se añade una restricción de precedencia, y es que las tareas dentro de un determinado trabajo tienen un cierto orden y deben ejecutarse de forma secuencial. El problema consiste en encontrar un orden de tareas para cada máquina que optimice una cierta función objetivo.

- **Flow Shop Scheduling Problem** (FSSP): es un GSSP en el que cada trabajo J_i consiste en M operaciones $\{\theta_{i1}, \ldots, \theta_{iM}\}$ con tiempos de procesamiento $p_{\theta_{ij}}$. En este caso existen restricciones de precedencia, y consisten en que las tareas de todos los trabajos deben utilizar las máquinas en orden secuencial, es decir la primera tarea de todos los trabajos se debe ejecutar en la máquina 1, la segunda tarea en la máquina 2, y así sucesivamente. Entonces, el problema consiste en encontrar un orden de trabajos tal que optimice una determinada función objetivo.

- **One Machine Sequence Problem** (OMSP): es un GSSP en el que cada trabajo J_i consiste en una única operación θ_i con tiempo de procesamiento p_i y que debe ser procesada sobre la máquina R. Es decir, cada trabajo tiene una única tarea, y existe una única máquina R para procesar todas las tareas. El problema consiste en encontrar el mejor orden de procesamiento de los trabajos J_i, sobre la máquina R.

El Job Shop Scheduling es el tipo de problemas elegido para esta tesis. En concreto, consideraremos una variante que incluye tiempos de setup dependientes de la secuencia. En la siguiente sección mostramos su descripción detallada.

2.3. Job Shop Scheduling Problem with Sequence-Dependent Setup Times

El SDST-JSP consiste en planificar un conjunto de N trabajos $\{J_1, \ldots, J_N\}$ en un conjunto R de M recursos físicos o máquinas $\{R_1, \ldots, R_M\}$. Cada trabajo J_i consiste en un conjunto de tareas u operaciones $\{\theta_{i1}, \ldots, \theta_{iM}\}$ que deben ser planificadas de forma secuencial, cada una de ellas en una máquina diferente. Cada tarea θ_{ij} requiere un único recurso, tiene una duración fija $p_{\theta_{ij}}$ y un tiempo de inicio $st_{\theta_{ij}}$ cuyo valor se debe determinar. Los trabajos pueden tener también un tiempo máximo de finalización d_j, y un peso w_j. Estos dos últimos valores no se utilizarán para minimizar el makespan, pero sí para otras funciones objetivo como el maximum lateness o el weighted tardiness.

Después de que una operación θ_{ij} salga de una máquina, y antes de que una operación θ_{kl} entre en la misma máquina, se debe realizar una operación de setup de duración $S_{\theta_{ij}\theta_{kl}}$. Por otra parte, $S_{0\theta_{ij}}$ es el tiempo de setup necesario antes de θ_{ij} si esta operación es la primera planificada en esa máquina, y de forma análoga, $S_{\theta_{ij}0}$ es el tiempo de limpieza después de la operación θ_{ij} si dicha operación es la última planificada en esa máquina.

A veces hablaremos de la desigualdad triangular de los tiempos de setup, y eso significará que se cumple la desigualdad $S_{uw} \leq S_{uv}+S_{vw}$, siendo u, v y w cualquier terna de operaciones que utilizan la misma máquina. Esta propiedad es aceptada como razonable por la mayoría de los autores. Nosotros la hemos utilizado en la demostración de algunas propiedades formales, y haremos mención explícita de ella siempre que así sea. Sin embargo, en la medida de lo posible diseñaremos estrategias cuyo funcionamiento sea independiente de su cumplimiento, ya que aunque es aplicable en la mayoría de los problemas reales, no tiene por qué cumplirse en todos ellos y de hecho no se cumplirá en varios de los benchmarks que utilizaremos.

El SDST-JSP tiene dos restricciones binarias: restricciones de precedencia y restricciones de capacidad. Las restricciones de precedencia están definidas por secuencialidad de las tareas dentro de un mismo trabajo, y se traducen en inecuaciones lineales del tipo: $st_{\theta_{ij}} + p_{\theta_{ij}} \leq st_{\theta_{i(j+1)}}$ (es decir, θ_{ij} antes que $\theta_{i(j+1)}$).

Las restricciones de capacidad limitan la utilización de cada recurso a una sola tarea de cada vez, y se traducen en restricciones disyuntivas de la forma: $st_{\theta_{ij}} + p_{\theta_{ij}} + S_{\theta_{ij}\theta_{kl}} \leq st_{\theta_{kl}} \vee st_{\theta_{kl}} + p_{\theta_{kl}} + S_{\theta_{kl}\theta_{ij}} \leq st_{\theta_{ij}}$, en donde θ_{ij} y θ_{kl} son operaciones que requieren la misma máquina.

2.4. Funciones objetivo para el SDST-JSP

A partir de esta definición del problema, se pueden proponer varias funciones objetivo diferentes. El objetivo más frecuente es obtener una planificación factible tal que el tiempo de fin de todos los trabajos, es decir, el *makespan*, denotado C_{max}, sea mínimo. Este problema entonces se denotaría por $J|s_{ij}|C_{max}$ según la notación $\alpha|\beta|\gamma$ utilizada en la literatura e introducida por Graham et al. en [65].

Para facilitar la notación a lo largo de esta tesis, si tenemos una tarea v vamos a denotar por PM_v a la tarea que se ejecuta inmediatamente antes que v en su misma máquina. De la misma forma denotaremos por SM_v a la tarea que se ejecuta inmediatamente después que v en su misma máquina. Por otra parte, denotaremos como PJ_v y SJ_v a la tarea inmedia-

tamente anterior o posterior, respectivamente, dentro del mismo trabajo de v. Entonces, si denotamos por C_v al tiempo de finalización de la tarea v, entonces el makespan (C_{max}) se define como:

$$C_{max} = \text{máx}\{C_v + S_{v0}\} \mid \forall v \text{ última tarea de su máquina} \tag{2.1}$$

Otro posible objetivo para el SDST-JSP puede ser obtener una planificación factible que haga mínimo el maximum lateness. En este caso, una instancia del problema también define un due date d_i para cada uno de los trabajos. d_i representa el tiempo máximo en que el trabajo i debería finalizar, lo que en un entorno real correspondería por ejemplo a un plazo máximo de entrega de un producto. Hay que tener en cuenta que un due date es una restricción blanda, en el sentido de que se puede violar a un cierto coste. Por el contrario un deadline sería una restricción dura, pero en esta tesis no los consideraremos. Si denotamos por C_i al tiempo de finalización del trabajo i, el maximum lateness de una planificación se define como

$$L_{max} = \max_{1 \leq i \leq N} \{C_i - d_i\} \tag{2.2}$$

Este problema se denota por $J|s_{ij}|L_{max}$ según la notación $\alpha|\beta|\gamma$.

Las dos funciones objetivo descritas pertenecen al tipo denominado *bottleneck objectives* según la clasificación definida por Brucker en [30]. Otros tipos de funciones objetivo son las *sum objectives* ([30]) los cuales se definen como la suma de funciones no decrecientes del tiempo de finalización de los trabajos, en lugar del máximo de dichas funciones. El total flow time o el weighted tardiness son funciones objetivo interesantes de este segundo tipo, y estos objetivos son más difíciles de resolver que los *bottleneck objectives*. Este hecho se observa claramente en los trabajos de Sierra et al. en [127], [128] y [129] a través de estudios experimentales con algoritmos exactos sobre las mismas instancias del JSP considerando tanto la minimización del makespan como la minimización del total flow time. Al mismo tiempo, las funciones objetivo como el weighted tardiness o el total flow time son más importantes que el makespan en muchos problemas del mundo real, pero a pesar de esto las investigaciones le han dedicado mucha más atención al makespan.

En el caso del weighted tardiness, para cada uno de los trabajos de la instancia se define un due date d_i y un peso w_i que indica la importancia del trabajo i. De nuevo denotaremos por C_i el tiempo en el que termina el trabajo i. Entonces el tardiness T_i del trabajo i se define como:

$$T_i = \text{máx}\{C_i - d_i, 0\} \tag{2.3}$$

El objetivo es minimizar el weighted tardiness total, es decir

$$\sum_{i=1,\ldots,N} w_i T_i \tag{2.4}$$

Utilizando la notación habitual para los problemas de planificación, este problema se denota por $J|s_{ij}|\sum w_i T_i$. Hay poca literatura disponible sobre esta función objetivo, a pesar de que en la vida real se considera más importante que la optimización del makespan, entre otras cosas porque mantener un buen servicio al cliente cada vez se considera más importante. Evitar la demora en la finalización de ciertos trabajos prioritarios puede ser mucho más importante que conseguir la minización del tiempo de finalización del último trabajo en ser ejecutado.

El total flow time se define como la suma de los tiempos de finalización de todos los trabajos, es decir, se pretende minimizar

$$\sum_{i=1,\ldots,N} C_i \tag{2.5}$$

Esto es un caso particular del weighted tardiness en el que $d_i = 0\ \forall i$ y $w_i = 1\ \forall i$. Este problema se denota por $J|s_{ij}|\sum C_i$. Al igual que ocurre con el weighted tardiness, también hay poca literatura sobre esta función con respecto a otras funciones objetivo como el makespan.

2.5. La representación mediante el modelo del grafo disyuntivo

El grafo disyuntivo es un modelo de representación frecuente en los problemas de scheduling. Fue propuesto por Roy y Sussmann en el año 1964 en [123] y a partir de entonces ha sido muy utilizado especialmente, pero no exclusivamente, para diseñar estrategias de resolución basadas en búsquedas por entornos. La definición de dicho grafo depende del problema en particular, e incluso de la función objetivo que deseemos minimizar. En esta tesis utilizaremos una representación ligeramente diferente para cada una de las cuatro funciones objetivo estudiadas. En las subsecciones siguientes se definen los grafos disyuntivos para cada una de ellas.

2.5.1. Minimización del makespan

Para la minimización del makespan en el SDST-JSP definimos el grafo disyuntivo como sigue. Una instancia del problema será un grafo dirigido $G = (V, A \cup E \cup I)$. Cada nodo del conjunto V representa una operación del problema, con la excepción de dos nodos *start* o 0, y *end* o $NM+1$, que representan operaciones ficticias con tiempo de procesamiento 0. Los arcos del conjunto A se llaman *arcos conjuntivos* y representan restricciones de precedencia, y los arcos del conjunto E se llaman *arcos disyuntivos* y representan restricciones de capacidad. Hay un arco del conjunto A entre cada operación v y su sucesora en el trabajo, es decir, $(v, SJ_v) \in A$, $\forall v$. El conjunto E se particiona en subconjuntos E_i, con $E = \cup_{i=1,\ldots,M} E_i$. E_i corresponde al recurso R_i e incluye un arco (v, w) para cada par ordenado de operaciones que requieren dicho recurso. Cada arco (v, w) de A tiene un coste, que es el tiempo de procesamiento de la operación del nodo origen, p_v, y cada arco (v, w) de E tiene un coste $p_v + S_{vw}$. El conjunto I incluye arcos de la forma $(start, v)$ y (v, end) para cada operación v del problema. Estos arcos tienen un coste S_{0v} y $p_v + S_{v0}$ respectivamente.

Una planificación factible se puede definir como una elección de los tiempos de inicio de todas las tareas del problema tal que cumpla con todas las restricciones que impone dicho problema, y se puede representar mediante un subgrafo acíclico G_s de G, $G_s = (V, A \cup H \cup J)$, donde $H = \cup_{i=1\ldots M} H_i$, siendo H_i una selección Hamiltoniana de E_i, y tal que J incluye arcos $(start, v_i)$ y (w_i, end) para todo $i = 1 \ldots M$, donde v_i y w_i son la primera y última operación de H_i respectivamente. Este grafo recibe el nombre de *grafo solución*.

Entonces, encontrar una solución se puede reducir a descubrir selecciones Hamiltonianas compatibles, es decir, órdenes de procesamiento para las operaciones que requieren el mismo recurso (a lo que también se puede denominar planificaciones parciales), y que se traducen en un grafo solución G_s acíclico. El makespan de la planificación es el coste de un *camino crítico*. Un *camino crítico* es el camino dirigido más largo posible desde el nodo *start* hasta el nodo *end*. Con camino más largo posible nos referimos a que la suma de los costes de todos los arcos que componen dicho camino debe ser la máxima posible. Pueden existir varios caminos críticos dentro de una misma planificación. Los nodos y arcos pertenecientes a un camino crítico se denominan *críticos*. El camino crítico se puede descomponer en subsecuencias $B_1, \ldots, B_r$ llamadas *bloques críticos*. Un *bloque crítico* es una subsecuencia maximal de operaciones en un camino crítico que requieren la misma máquina. Nótese que, obviamente, un bloque crítico puede estar formado por una única tarea si la anterior y la siguiente a ella en el camino crítico, si existen, son de su mismo trabajo.

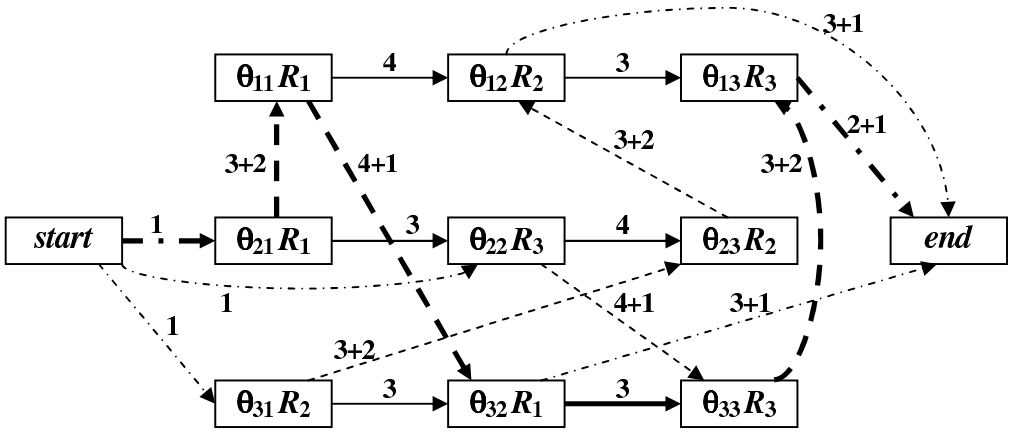


Figura 2.1: Modelo de grafo disyuntivo para el makespan

La figura 2.1 muestra una planificación factible para una instancia con 3 trabajos y 3 máquinas. Las flechas discontínuas representan los elementos de H, las flechas contínuas los elementos de A y las flechas mixtas (puntos seguidos de rayas) los elementos de J. Los arcos en negrita muestran un camino crítico cuya longitud, es decir el makespan, es 22.

A partir del grafo solución es fácil asignar tiempos de inicio a las tareas: a partir del instante 0 cada una comienza tan pronto como finalicen todas sus predecesoras. Esto sería la planificación natural que representa el orden de procesamiento de las tareas que codifica el digrafo solución, y será la estrategia básica de un constructor de planificaciones semiactivas, tal y como se explicará en la sección 3. Por otra parte, observando el grafo solución también se puede saber si la solución que representa es factible o no. La solución será factible sólo si no existen ciclos en el grafo.

Para simplificar las expresiones, definiremos la siguiente notación. La *cabeza* r_v de una operación v es el coste del camino más largo desde el nodo *start* hasta el nodo v en el grafo solución, es decir, será el tiempo de inicio de la tarea v. La *cola* q_v esta definida tal que el valor $q_v + p_v$ es el coste del camino más largo desde el nodo v hasta el nodo *end*. Por lo tanto, $r_v + p_v + q_v$ es la longitud del camino más largo desde el nodo *start* hasta el nodo *end* que pasa a través del nodo v, y por lo tanto es una cota inferior del makespan. Además, será el makespan si el nodo v pertenece a un camino crítico.

Vamos a aclarar los conceptos de cabeza y cola con el ejemplo de dos tareas de la planificación de la figura 2.1.

Comenzaremos con la tarea crítica θ_{32}: $r_{\theta_{32}} = 11$, ya que podemos ver que el camino más largo desde el nodo *start* hasta θ_{32} pasa por las tareas θ_{21} y θ_{11}. De la misma forma, $q_{\theta_{32}} = 8$, debido a que el camino más largo desde θ_{32} hasta el nodo *end* pasa por las tareas θ_{33} y θ_{13}. Además, vemos que $r_{\theta_{32}} + p_{\theta_{32}} + q_{\theta_{32}} = 11 + 3 + 8 = 22$, que es igual que el makespan de la planificación, y esto confirma que θ_{32} pertenece a un camino crítico.

Ahora consideremos la tarea no crítica θ_{22}: $r_{\theta_{22}} = 4$, porque el camino más largo desde el nodo *start* hasta θ_{22} pasa por la tarea θ_{21}. Por otra parte, $q_{\theta_{22}} = 11$, ya que el camino más largo desde θ_{22} hasta el nodo *end* pasa por las tareas θ_{23}, θ_{12} y θ_{13}. Por último, tenemos que $r_{\theta_{22}} + p_{\theta_{22}} + q_{\theta_{22}} = 4 + 4 + 11 = 19$, que es menor al makespan de la planificación, y esto confirma que θ_{22} no pertenece a un camino crítico.

A continuación definiremos más formalmente las cabezas y las colas. Recordemos que denotamos por PJ_v y SJ_v al predecesor y sucesor respectivamente de v en la secuencia de su trabajo, y por PM_v y SM_v al predecesor y sucesor de v en la secuencia de su máquina. Por razones prácticas, consideraremos el nodo *start* como PJ_v para la primera operación de cada trabajo y como PM_v para la primera operación ejecutada en cada máquina; y además $p_{start} = 0$. También consideraremos el nodo *end* como SJ_v para la última operación de cada trabajo y como SM_v para la última operación de cada máquina, y además que $p_{end} = 0$.

Entonces, la cabeza de cada operación v y de cada tarea ficticia del grafo se puede calcular como sigue:

$$r_{start} = 0$$

$$r_v = \text{máx}(r_{PJ_v} + p_{PJ_v}, r_{PM_v} + p_{PM_v} + S_{PM_v v})$$

$$r_{end} = \max_{v \in PM_{end}} \{r_v + p_v + S_{v0}\}$$

Y la cola de cada operación v y de cada tarea ficticia del grafo se puede calcular de la siguiente forma:

$$q_{end} = 0$$

$$q_v = \text{máx}(q_{SJ_v} + p_{SJ_v}, q_{SM_v} + p_{SM_v} + S_{vSM_v})$$

$$q_{start} = \max_{v \in SM_{start}} \{q_v + p_v + S_{0v}\}$$

Entonces, una operación v se puede planificar si tanto PJ_v como PM_v ya se han planificado, y además se ha completado el tiempo de setup correspondiente $S_{PM_v v}$. El tiempo de inicio

para cada tarea se determina por el máximo entre el tiempo de fin de su predecesora en el trabajo y el tiempo de fin de su predecesora en el recurso sumado con el tiempo de setup. Formalmente:

$$C_v = r_v + p_v$$
$$r_v \geq \text{máx}(C_{PJ_v}, C_{PM_v} + S_{PM_v v})$$

2.5.2. Minimización del maximum lateness

Como ya hemos adelantado, para abordar la minimización de otras funciones objetivo en el SDST-JSP definiremos un modelo de grafo disyuntivo diferente para cada una de ellas. En el caso de la minimización del maximum lateness la definición del grafo será muy similar a la utilizada para la minimización del makespan, sin embargo hay algunas diferencias que detallaremos. A continuación proponemos la siguiente definición formal.

Una instancia del problema $J|s_{ij}|L_{max}$ se puede representar mediante un grafo dirigido $G = (V, A \cup E \cup I_1 \cup I_2 \cup I_3)$. Cada nodo del conjunto V representa una operación del problema, con la excepción de los nodos dummy *start*, end_i $1 \leq i \leq N$ y *end* que representan operaciones ficticias con tiempo de procesamiento nulo y que no requieren ninguna máquina, y se utilizan para darle a G una cierta estructura. Al igual que en el grafo utilizado para minimizar el makespan, los arcos del conjunto A se denominan *arcos conjuntivos* y representan restricciones de precedencia, los arcos de E se denominan *arcos disyuntivos* y representan restricciones de capacidad, y el conjunto E se particiona en subconjuntos E_i, con $E = \cup_{i=1,\ldots,M} E_i$, en donde E_i corresponde al recurso R_i e incluye un arco (v, w) para cada par ordenado de operaciones que requieren dicho recurso. Cada arco (v, w) de A tiene como coste el tiempo de procesamiento de la operación de su nodo origen, p_v, y cada arco (v, w) de E tiene como coste $p_v + S_{vw}$. El conjunto I_1 incluye arcos de la forma $(start, v)$ para cada operación v del problema. Estos arcos tienen como coste S_{0v}. El conjunto I_2 incluye arcos (θ_{iM}, end_i), $1 \leq i \leq N$, que tienen como coste $p_{\theta_{iM}}$. Por último, los arcos de I_3 son de la forma (end_i, end) y tienen como coste $-d_i$.

Con esta definición del grafo no estamos considerando los tiempos de setup de limpieza final de las máquinas. Esto es así debido a que al minimizar el maximum lateness no es necesario tener en cuenta la duración de la planificación completa, como ocurría con el makespan, sino exclusivamente los tiempos de finalización de cada trabajo en particular.

La principal diferencia respecto al grafo definido para el makespan es el añadido de los

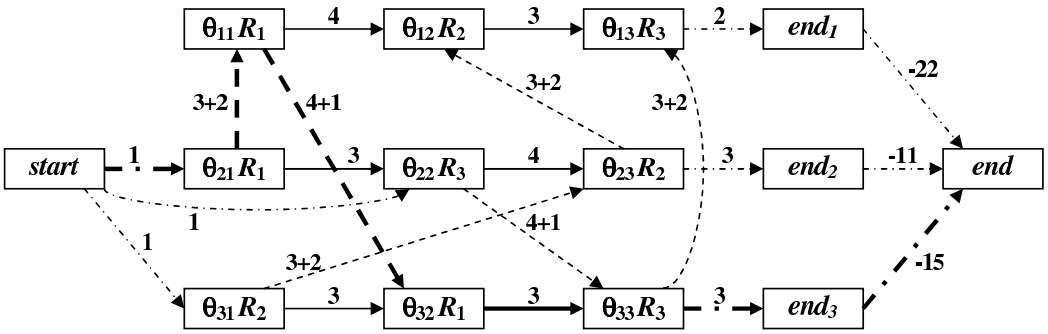


Figura 2.2: Modelo de grafo disyuntivo para el maximum lateness

nodos finales end_i, $1 \leq i \leq N$. Este modelo de grafo para el problema $J|s_{ij}|L_{max}$ es una extensión natural de los modelos de grafo disyuntivo propuestos por Mehta et al. en [90] o por Vela et al. en [151] para los problemas $J||L_{max}$ y $J|s_{ij}|C_{max}$, respectivamente.

De forma similar a la minimización del makespan, encontrar una planificación factible se reduce a descubrir órdenes de procesamiento para las operaciones que utilizan el mismo recurso tales que formen un subgrafo acíclico G_s de G, $G_s = (V, A \cup H \cup J_1 \cup I_2 \cup I_3)$, donde $H = \cup_{i=1...M} H_j$, H_j es una selección Hamiltoniana de E_j. J_1 incluye arcos $(start, v_j)$ para todos los $j = 1 \ldots M$, siendo v_j la primera operación de H_j.

La figura 2.2 muestra una planificación factible para una instancia con 3 trabajos y 3 máquinas. Las flechas discontínuas representan los elementos de H, las flechas contínuas los elementos de A y las flechas mixtas (puntos seguidos de rayas) los elementos de J_1, de I_2 y de I_3. Los arcos en negrita muestran un camino crítico, cuya longitud, es decir el maximum lateness, es 2.

A partir de la definición de G_s, el maximum lateness de la planificación que representa el orden de procesamiento de las tareas codificadas en G_s es el coste de un *camino crítico* en G_s. Con la definición propuesta para el grafo disyuntivo, la definición de caminos críticos, bloques críticos, y formas de construir una planificación son idénticas a las descritas en la sección de minimización del makespan. Sin embargo, la forma de calcular cabezas y colas difiere ligeramente, como se verá a continuación.

La cabeza de una operación v, denotada por r_v, es el coste del camino más largo desde el nodo *start* hasta el nodo v y representa el tiempo de inicio de la tarea v en la planificación que representa G_s. La cola q_v es el coste del camino más largo desde el nodo v hasta el

nodo *end*, menos la duración de la tarea del nodo v, y representa el maximum lateness del subproblema dado por el nodo v junto con todos sus sucesores en el orden de procesamiento dado por G_s. Denotaremos por J_v el trabajo de la operación v. Recordemos que consideramos el nodo *start* como PJ_v para la primera operación de cada trabajo y como PM_v para la primera operación ejecutada en cada máquina; y que $p_{start} = 0$. Entonces, la cabeza de cada operación v y de cada tarea ficticia del grafo se puede calcular como sigue:

$$r_{start} = 0$$

$$r_v = \text{máx}(r_{PJ_v} + p_{PJ_v}, r_{PM_v} + p_{PM_v} + S_{PM_v v})$$

$$r_{end_i} = r_v + p_v,\ (v, end_i) \in I_2, 1 \leq i \leq N$$

$$r_{end} = \max_{1 \leq i \leq N} \{r_{end_i} - d_i\}$$

Ahora, para cada $1 \leq i \leq N$, consideramos el nodo *end* como SJ_{end_i} para cada trabajo i, el nodo end_i como SJ_v para la última tarea del trabajo i, y tenemos además que $p_{end_i} = -d_i$ y que $p_{end} = 0$. Entonces, la cola de cada operación v y de cada tarea ficticia del grafo se puede calcular como sigue:

$$q_{end} = q_{end_i} = 0$$

$$q_v = \begin{cases} \text{máx}(q_{SJ_v} + p_{SJ_v}, q_{SM_v} + p_{SM_v} + S_{vSM_v}) & \text{si } SM_v \text{ existe} \\ q_{SJ_v} + p_{SJ_v} & \text{en caso contrario} \end{cases}$$

$$q_{start} = \max_{v \in SM_{start}} \{q_v + p_v + S_{0v}\}$$

Claramente, las cabezas deben calcularse desde el nodo *start* hasta el nodo *end*, y las colas desde el nodo *end* hasta el nodo *start*. Es fácil ver que un nodo v es crítico si y sólo si $L_{max} = r_v + p_v + q_v$.

Como conclusión, la representación propuesta para el grafo disyuntivo permite transformar el problema de minimizar el maximum lateness en un problema de minimizar el makespan, $J|s_{ij}|C_{max}$. De hecho en la representación propuesta, si consideramos los arcos (end_i, end) con un peso nulo en lugar de $-d_i$ y consideramos nulos los tiempos de setup de limpieza final de las máquinas, el grafo disyuntivo resultante es equivalente al grafo original definido para la minimización del makespan, es decir que la longitud de un camino crítico en G_s es el valor del makespan. Gracias a esto, para minimizar el maximum lateness en principio se podrán utilizar algunos resultados y algoritmos ya desarrollados para la minimización del makespan, lo cual es muy útil porque es una función objetivo mucho más estudiada. Sin

embargo, debemos tener muy en cuenta las diferencias en la formulación del problema: ahora tenemos M operaciones ficticias, $end_1 \dots end_M$, con duraciones negativas, $-d_1 \cdots - d_m$, y que no requieren ninguna máquina. Veremos que las estructuras de vecindad basadas en movimientos dentro del camino crítico definidas para el makespan podrán ser extendidas de forma natural para tratar el problema del maximum lateness.

2.5.3. Minimización del weighted tardiness

Para el caso del problema $J|s_{ij}|\sum w_i T_i$ proponemos la siguiente definición para el grafo disyuntivo. Será un grafo dirigido $G = (V, A \cup E \cup I_1 \cup I_2)$. Cada nodo del conjunto V representa una operación del problema, con la excepción de los nodos *start* y end_i $1 \leq i \leq N$, que representan operaciones ficticias que no requieren ninguna máquina y se utilizan para dotar al grafo de una cierta estructura. Los arcos de A se denominan *arcos conjuntivos* y representan las restricciones de precedencia, y los arcos de E se denominan *arcos disyuntivos* y representan las restricciones de capacidad. El conjunto E se particiona en subconjuntos E_i, con $E = \cup_{i=1,\dots,M} E_i$, donde E_i corresponde al recurso R_i e incluye un arco (v, w) para cada par ordenado de operaciones que requieren ese recurso. Cada arco (v, w) de A tiene un peso igual al tiempo de procesamiento que requiere la operación del nodo origen, p_v, y cada arco (v, w) de E tiene un peso igual a $p_v + S_{vw}$. El conjunto I_1 incluye arcos de la forma $(start, v)$ para cada operación v del problema. Estos arcos tienen como coste S_{0v}. El conjunto I_2 incluye arcos (θ_{iM}, end_i), $1 \leq i \leq N$, con un peso igual a $p_{\theta_{iM}}$.

Al igual que en el caso del maximum lateness, no se considerarán los tiempos de setup de limpieza final de las máquinas. En este caso la principal diferencia respecto al grafo definido para el makespan es la sustitución del único nodo final *end* por una serie de nodos finales end_i $1 \leq i \leq N$.

Una planificación factible se representa mediante un subgrafo acíclico G_s de G, $G_s = (V, A \cup H \cup J_1 \cup I_2)$, donde $H = \cup_{j=1\dots M} H_j$, siendo H_j una selección Hamiltoniana de E_j. J_1 incluye arcos $(start, v_j)$ para todos los $j = 1 \dots M$, siendo v_j la primera operación de H_j.

La figura 2.3 muestra una planificación factible para una instancia con 3 trabajos y 3 máquinas. Las flechas discontínuas representan los elementos de H, las flechas contínuas los elementos de A y las flechas mixtas (puntos seguidos de rayas) los elementos de J_1 y de I_2.

Si denotamos por C_i el tiempo en el que termina el trabajo i, dicho valor es el coste del camino dirigido en G_s desde el nodo *start* hasta el nodo end_i que tiene el mayor coste. Entonces, el weighted tardiness de la planificación es $\sum_{i=1,\dots,N} w_i T_i$, con $T_i = \text{máx}(C_i -$

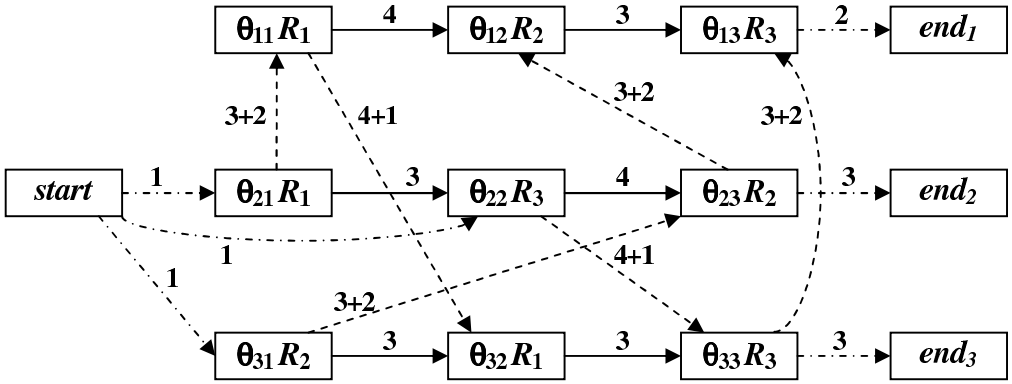


Figura 2.3: Modelo de grafo disyuntivo para el weighted tardiness y el total flow time

$d_i, 0$). La definición de camino crítico cambia ligeramente. Ahora un camino crítico es un camino dirigido en G_s desde el nodo *start* hasta un nodo end_i con el mayor coste posible, siempre y cuando $T_i > 0$ para el trabajo i. Es decir, sólo consideraremos como caminos críticos los que acaben en el nodo final de un trabajo que no cumpla su due date. Nótese que, a diferencia de las funciones objetivo anteriores, en este caso la longitud de un camino crítico no coincide con el valor del objetivo que deseamos optimizar. La definición de bloque crítico, sin embargo, es idéntica. En cuanto a las definiciones de cabezas y colas, la de cabeza será idéntica pero la de cola cambia significativamente, como se verá a continuación.

La cabeza de una operación v, denotada por r_v, es el coste del camino más largo desde el nodo *start* hasta el nodo v y corresponde al tiempo de inicio de la operación v en la planificación representada por G_s. La cola q_v^i, $1 \leq i \leq N$, es el coste del camino más largo desde el nodo v hasta el nodo end_i, menos la duración de la tarea del nodo v. Es fácil ver que un nodo v es critico si y sólo si $r_v + p_v + q_v^j = C_j$ para algún trabajo j. Por razones prácticas, vamos a considerar $q_v^i = -\infty$ cuando no existe un camino desde v hasta end_i. Es importante remarcar que hemos definido N colas para cada operación, mientras que para la minimización del makespan o del maximum lateness únicamente se necesita definir una cola.

De nuevo vamos a considerar el nodo *start* como PJ_v cuando v es la primera tarea de su trabajo y como PM_v cuando v es la primera tarea que será ejecutada en una máquina.

Entonces, las cabezas de cada operación v y de cada nodo ficticio del grafo se pueden calcular como sigue:

$$r_{start} = 0$$
$$r_v = \text{máx}(r_{PJ_v} + p_{PJ_v}, r_{PM_v} + p_{PM_v} + S_{PM_v v})$$
$$r_{end_i} = r_v + p_v,\ (v, end_i) \in I_2, 1 \leq i \leq N$$

Ahora, consideramos el nodo end_i, $1 \leq i \leq N$, como SJ_v cuando v es la última tarea del trabajo i, y tenemos además que $p_{end_i} = 0$. Entonces, la cola de cada operación v y de cada tarea ficticia del grafo se puede calcular como sigue:

$$q^i_{end_i} = 0$$
$$q^j_{end_i} = -\infty, j \neq i$$
$$q^j_v = \text{máx}(q^j_{SJ_v} + p_{SJ_v}, q^j_{SM_v} + p_{SM_v} + S_{vSM_v}) \quad \text{si } SM_v \text{ existe}$$
$$q^j_v = q^j_{SJ_v} + p_{SJ_v} \quad \text{en caso contrario}$$
$$q^j_{start} = \underset{v \in SM_{start}}{\text{máx}} \{q^j_v + p_v + S_{0v}\}$$

Se puede ver que las cabezas se deben calcular desde el nodo *start* en adelante, mientras que las colas se deben calcular desde los nodos end_i hacia atrás. En conclusión, la minimización del weighted tardiness nos ha obligado a definir N nodos finales en el grafo disyuntivo, y a cambiar las definiciones de camino crítico y de cola de una tarea. Ahora habrá una mayor cantidad de caminos críticos si hay muchos trabajos que no cumplan su due date, y además cada tarea tendra N colas, siendo N el número total de trabajos de la instancia.

2.5.4. Minimización del total flow time

El grafo disyuntivo utilizado para la minimización del total flow time será idéntico al definido anteriormente para el weighted tardiness. Esto es así porque el total flow time es un caso especial del weighted tardiness, como ya se ha explicado anteriormente. La definición de cabezas y colas será también idéntica.

Sin embargo, hay que tener en cuenta que en el caso del total flow time se puede considerar que ningún trabajo cumple su due date (ya que todos los due dates son 0). Por lo tanto, utilizando la misma definición de camino crítico utilizada para el weighted tardiness, ahora

se considerará crítico un camino en G_s desde el nodo *start* hasta cualquier nodo end_i con el mayor coste posible. Es decir, habrá como mínimo un camino crítico por cada trabajo que tenga la instancia, lo cual no tiene por qué ocurrir en la minimización del weighted tardiness.

2.6. Diagramas de Gantt y otras formas de representación

La forma más típica de representar soluciones para problemas de planificación de tareas es el diagrama de Gantt. En estos diagramas el eje X representa el tiempo y el eje Y el recurso. Dentro del diagrama hay un recuadro por cada tarea de cada trabajo. Dicho recuadro se sitúa en la fila que corresponde a su recurso, y en el eje X abarca todo el tiempo comprendido desde el inicio de la ejecución de dicha tarea hasta su final. Dentro de dichos recuadros se puede introducir algo de información adicional, como la identificación de la tarea. Otra alternativa sería un diagrama de Gantt que utiliza el eje Y para indicar el trabajo en lugar del recurso.

Como ejemplo, la figura 2.4 muestra la solución representada en el grafo de la figura 2.1 mediante un diagrama de Gantt por recursos, es decir, un diagrama de Gantt en el que el eje Y indica el recurso. Se han coloreado las tareas de diferente forma según el trabajo al que pertenecen, y los tiempos de setup se colorean en gris y los hemos denotado de forma resumida mediante una S.

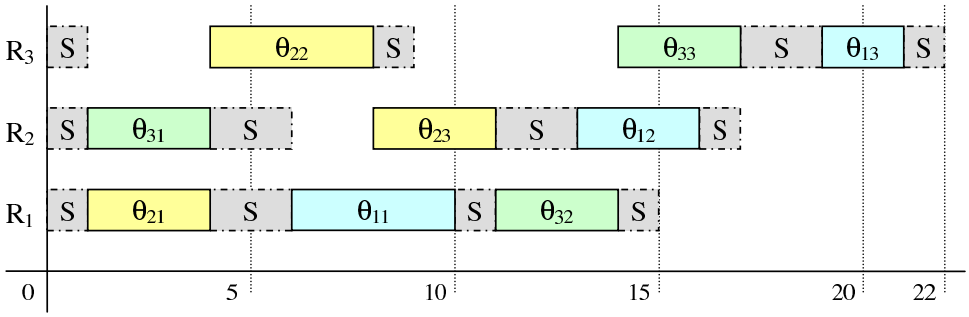


Figura 2.4: Diagrama de Gantt para la planificación representada en la figura 2.1

Otra forma de representar una solución de un problema de scheduling es mediante una matriz, en la que el número de fila representa el número de recurso, mientras que el número de columna es el orden de planificación dentro de dicho recurso. Esta representación es mucho

más simple que el diagrama de Gantt, pero tiene el inconveniente de que no incluye ninguna información sobre tiempos de inicio y finalización de las tareas. Observar que la matriz se puede deducir de forma trivial a partir del diagrama de Gantt. Por ejemplo, la solución representada por el anterior diagrama de Gantt correspondería a la siguiente matriz:

$$\begin{pmatrix} \theta_{21} & \theta_{11} & \theta_{32} \\ \theta_{31} & \theta_{23} & \theta_{12} \\ \theta_{22} & \theta_{33} & \theta_{13} \end{pmatrix}$$

Otra representación posible de una solución es mediante un vector que contendrá el orden de procesamiento de todas las tareas. Este orden puede codificarse también mediante una permutación con repetición, y en este último caso se incluirá únicamente información del trabajo de la tarea correspondiente. En concreto, la primera aparición del identificador de un trabajo hace referencia a la primera tarea de dicho trabajo, la segunda vez que aparece hace referencia a la segunda tarea del trabajo, y así sucesivamente. Por ejemplo, si se nombra el primer trabajo como θ_1, el segundo trabajo como θ_2 y el tercer trabajo como θ_3, una representación de la solución de la matriz anterior podría ser la siguiente:

$$\begin{pmatrix} \theta_2 & \theta_1 & \theta_3 & \theta_2 & \theta_3 & \theta_2 & \theta_1 & \theta_3 & \theta_1 \end{pmatrix}$$

Mientras que una permutación convencional que codifique directamente el orden de procesamiento de las tareas es la siguiente:

$$\begin{pmatrix} \theta_{21} & \theta_{11} & \theta_{31} & \theta_{22} & \theta_{32} & \theta_{23} & \theta_{12} & \theta_{33} & \theta_{13} \end{pmatrix}$$

Utilizando este tipo de representación puede suceder que dos permutaciones distintas representen la misma planificación. Nótese que una permutación no es sino un orden topológico del grafo solución, y obviamente el orden relativo entre dos tareas que no comparten recurso ni trabajo es irrelevante, por lo que aunque aparezcan permutadas codifican un mismo grafo. A continuación mostramos dos ejemplos de permutaciones diferentes de las de los ejemplos anteriores pero que representan la misma solución, el primero de ellos utilizando permutaciones con repetición, y el segundo permutaciones convencionales:

$$\begin{pmatrix} \theta_3 & \theta_2 & \theta_2 & \theta_1 & \theta_3 & \theta_3 & \theta_2 & \theta_1 & \theta_1 \end{pmatrix}$$

$$\begin{pmatrix} \theta_{31} & \theta_{21} & \theta_{22} & \theta_{11} & \theta_{32} & \theta_{33} & \theta_{23} & \theta_{12} & \theta_{13} \end{pmatrix}$$

La gran ventaja de las permutaciones con repetición es que siempre dan lugar a una planificación factible. Sin embargo en la utilización de permutaciones convencionales debemos tener en cuenta que en los problemas de tipo job shop las tareas dentro de cada trabajo deben seguir un cierto orden, y por este motivo se pueden generar soluciones no factibles si no se tiene cuidado al construir la permutación.

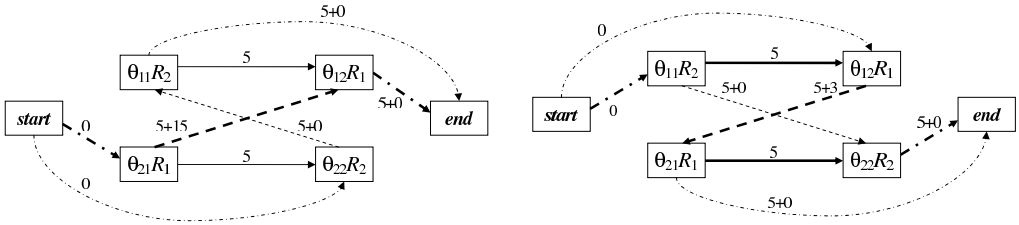
2.7. Caminos críticos y soluciones óptimas

Los conceptos de camino crítico y bloque crítico son de gran importancia para los problemas de planificación, ya que la mayoría de las propiedades formales y métodos de resolución se basan en ellos. Por ejemplo, es bien sabido que en la minimización del makespan en una instancia del JSP, una planificación es óptima si un camino crítico se compone únicamente de operaciones del mismo trabajo u operaciones de la misma máquina. Esta es una propiedad interesante, ya que se puede utilizar como criterio de parada en un algoritmo de búsqueda. Sin embargo para el SDST-JSP éste no es el caso, como prueban los dos teoremas presentados en esta sección.

Estos teoremas se pueden aplicar tanto a la minimización del makespan como a la minimización del maximum lateness en el SDST-JSP (ya que con el grafo disyuntivo propuesto estas dos funciones son equivalentes, exceptuando los tiempos finales de setup de limpieza). En la minimización de las funciones objetivo de tipo suma como el weighted tardiness o el total flow time, estos criterios no tienen tanta utilidad como condiciones de parada, debido a que esas funciones tienen varios caminos críticos que acaban en nodos finales distintos. Podría ser interesante, para las funciones objetivo de tipo suma, probar que un camino hasta un determinado nodo final es óptimo, pero ese resultado tampoco tendría mucha utilidad ya que puede ocurrir que la planificación óptima tenga ese camino más largo si con ello permite que otros caminos hacia otros nodos finales sean más cortos.

Teorema 2.1. *Sea H una planificación de una instancia SDST-JSP. Incluso aunque H tenga un camino crítico tal que todas sus operaciones sean de la misma máquina, H puede no ser óptimo.*

Demostración. Demostraremos este resultado mediante un contraejemplo. Consideremos una instancia con 2 trabajos y 2 máquinas. θ_{11} y θ_{22} requieren R_1, y θ_{12} y θ_{21} requieren R_2. El tiempo de procesamiento para todas las operaciones es de 5. Todos los tiempos de setup son nulos, excepto $S_{23} = 3$ y $S_{32} = 15$. Esta claro que la desigualdad triangular de



(a) Planificación con makespan 25 (b) Solución óptima para el mismo problema

Figura 2.5: Una planificación cuyo camino crítico se compone únicamente de operaciones de la misma máquina puede no ser óptima

tiempos de setup se cumple para esta instancia, por lo que este resultado es válido aún bajo la hipótesis de que se cumpla esta desigualdad. La figura 2.5 ilustra este contraejemplo. (a) muestra una planificación que no es óptima, con un camino crítico que consiste únicamente en dos operaciones que requieren la misma máquina y un makespan de 25, y (b) muestra la planificación óptima para la misma instancia con un makespan de 23.

□

Teorema 2.2. *Sea H una planificación de una instancia SDST-JSP con un camino crítico tal que todas sus operaciones pertecenen al mismo trabajo. Si se cumple la desigualdad triangular de tiempos de setup, entonces H es óptima.*

Demostración. Sea i el trabajo cuyas operaciones forman el camino crítico. Entonces la longitud de dicho camino es $S_{0\theta_{i1}} + \sum_{j=1,\dots,M} p_{\theta_{ij}} + S_{\theta_{iM}0}$. Debido a la desigualdad triangular, θ_{i1} no puede empezar antes que $S_{0\theta_{i1}}$ y el tiempo después de θ_{iM} no puede ser más pequeño que $S_{\theta_{iM}0}$, por lo tanto la longitud de ese camino crítico es una cota inferior de la planificación óptima, y por lo tanto H es óptima. □

Vemos que el teorema anterior requiere que se cumpla la desigualdad triangular de tiempos de setup. En caso de que no se cumpla también se puede probar la optimalidad, pero bajo ciertas condiciones. Vemos que la única forma de reducir la longitud de ese camino crítico es llegar por otro camino desde el nodo inicial hasta la primera tarea del trabajo i en un tiempo menor que $S_{0\theta_{i1}}$ (o de forma equivalente, llegar desde la última tarea del trabajo i hasta el nodo final en un tiempo menor que $S_{\theta_{iM}0}$). Es obvio que el camino alternativo más corto posible tiene que pasar por, como mínimo, una tarea de otro trabajo. Por lo tanto para que el camino alternativo tenga una longitud menor, $S_{0\theta_{i1}}$ debería ser mayor que la suma de como mínimo tres valores: otro tiempo de setup inicial, la duración de la primera tarea

de otro trabajo, y el tiempo de setup necesario para pasar de dicha tarea a la primera tarea del trabajo i. El mismo razonamiento se aplicaría para el camino desde la última tarea del trabajo i hasta el nodo final. Teniendo esto en cuenta, se pueden pensar varias condiciones bajo las que se puede probar la optimalidad, y a continuación mostramos varios ejemplos:

- Si los tiempos de setup iniciales o finales de la instancia son siempre menores que las duraciones de las tareas de la instancia.
- Si la instancia no tiene tiempos de setup cuya duración sea menos de la mitad que otro tiempo de setup.
- Si $S_{0\theta_{i1}}$ es el mayor tiempo de setup inicial de la instancia.
- Si la primera tarea del trabajo i no comparte máquina con la primera tarea de ningún otro trabajo, y los tiempos de setup iniciales de la instancia son siempre menores que la suma de las duraciones de dos cualesquiera de las tareas de la instancia (ya que en este caso el camino alternativo más corto debe pasar por al menos dos tareas).

Sin embargo todas estas condiciones son específicas de cada instancia en particular y por lo tanto su aplicación es más compleja, por lo que no las tendremos en cuenta en la experimentación.

2.8. Conclusiones

En este capítulo hemos realizado una breve introducción a los problemas de scheduling. Hemos definido el problema de scheduling general llamado General Shop Scheduling Problem (GSSP) y varios casos particulares de dicho problema, entre los cuales se encuentra el JSP clásico. A continuación hemos descrito formalmente el SDST-JSP, que es un caso más general del JSP que considera tiempos de mantenimiento entre operaciones consecutivas en la misma máquina. Después definimos todas las funciones objetivo que se utilizarán a lo largo de esta tesis: makespan, maximum lateness, weighted tardiness y total flow time. El modelo del grafo disyuntivo es una de las formas más utilizadas de representar problemas de scheduling, y en este trabajo también será el modelo utilizado, aunque con ligeras diferencias según la función objetivo que deseemos optimizar. Este grafo además permite evaluar cuándo una posible solución del problema es factible o no, ya que si existe algún ciclo en el grafo la planificación no será factible. Hemos introducido también los diagramas de Gantt, que son

una forma muy práctica y visual de representar planificaciones o soluciones del problema. Además, también hemos presentado varias definiciones muy importantes que se utilizarán a lo largo de esta tesis, como son la de camino crítico, bloque crítico, cabezas o colas, y hemos demostrado que algunas de las propiedades que cumplen los caminos críticos en el JSP clásico dejan de cumplirse en el SDST-JSP, lo que dificulta su resolución.

Capítulo 3

TIPOS DE PLANIFICACIONES Y CONSTRUCTORES DE PLANIFICACIONES

3.1. Introducción

El espacio completo de planificaciones factibles es muy grande y además contiene muchas soluciones que no son realmente interesantes. Por ello se han buscado espacios de búsqueda alternativos con un tamaño mucho más pequeño, pero con la condición de que sean completos, es decir que contengan al menos una solución óptima. Bajo estas condiciones existen dos espacios de búsqueda de interés: los espacios de planificaciones semiactivas y activas.

Una planificación es *semiactiva* si para adelantar el tiempo de inicio de una tarea en una máquina, es necesario modificar el orden de al menos dos tareas.

Una planificación es *activa* si es imprescindible que el inicio de una operación deba ser retrasado si queremos que otra pueda empezar antes.

El conjunto de las planificaciones activas es un subconjunto de las planificaciones semiactivas, que a su vez es un subconjunto de las planificaciones factibles. El interés de utilizar planificaciones activas en un problema radica en que es un espacio de búsqueda mucho más pequeño, y además se sabe que incluye al menos una planificación óptima, es decir, es un

conjunto dominante.

La experiencia muestra que el valor medio del makespan es mucho mayor para las planificaciones semiactivas que para las planificaciones activas. Pero también es cierto que los algoritmos que planifican de forma semiactiva tienen una complejidad menor que los algoritmos que planifican de forma activa. Por lo tanto cada tipo de planificación tiene sus ventajas y se deberá utilizar el tipo de planificación más adecuado en cada momento.

El proceso para planificar una solución de forma semiactiva es muy simple. Como ya explicamos en la sección 2.6, una solución a una instancia del problema se puede representar por un vector o por una matriz. Entonces, el algoritmo simplemente recorrería la solución de forma secuencial y planificaría las tareas que vayan apareciendo en el tiempo de inicio más temprano que sea posible, siempre a continuación de la última tarea planificada de la correspondiente máquina. Es trivial demostrar que este algoritmo simple de planificación es dominante, ya que el espacio de soluciones semiactivas contiene como mínimo una solución óptima, y cualquier planificación semiactiva se puede representar mediante una permutación o una matriz, sin más que ordenar las tareas en orden no decreciente de sus tiempos de inicio.

Las planificaciones activas son algo más complejas de construir, y en las siguientes secciones se explicarán varios métodos para generarlas.

Existe otro tipo básico de planificaciones denominadas non-delay o *densas*, que consisten en evitar que una máquina esté inactiva en un momento en el que podría estar ejecutando alguna tarea. Este tipo de planificaciones son útiles cuando se requiere encontrar una planificación aceptable en un tiempo muy reducido. Sin embargo la solución óptima no tiene por qué estar dentro de este conjunto, y por lo tanto en metaheurísticas de búsqueda no suele ser conveniente utilizar planificaciones densas, salvo que el tamaño del espacio de búsqueda sea muy grande.

3.2. Constructor de planificaciones activas G&T para el JSP

Para calcular planificaciones activas para el problema JSP clásico se suele utilizar el conocido algoritmo $G\&T$, propuesto por Giffler y Thompson en [55]. Éste es un algoritmo voraz que, en cada iteración, calcula un conjunto de tareas candidatas a ser planificadas, de modo que cualquiera que sea la tarea elegida dentro de este conjunto, el algoritmo garantiza que la planificación resultante es activa.

Existen diversas variantes de este algoritmo y algunas de ellas consisten en reducir el espacio de búsqueda. Esto puede ser útil, ya que el conjunto de planificaciones activas suele ser muy grande en problemas de gran tamaño. El problema de realizar esta reducción del espacio de búsqueda es que se corre el riesgo de perder todas las planificaciones óptimas.

En la variante más conocida del $G\&T$ existe un mecanismo de control del tiempo de inactividad máxima permitida a una máquina mediante un parámetro $\delta \in [0,1]$, y en este caso se suele llamar algoritmo $G\&T$ híbrido. Mostramos el código del $G\&T$ híbrido en el algoritmo 3.1.

Algoritmo 3.1 Constructor de planificaciones $G\&T$ híbrido para el JSP

Requiere: Un cromosoma C, una instancia de problema P, y un valor $\delta \in [0,1]$

Produce: Una planificación construída a partir del cromosoma C para la instancia P

1. $A = \{\theta_{i1}, 1 \leq i \leq N\}$;

mientras $A \neq \emptyset$ **hacer**

2. $\forall \theta \in A$ sea $st_\theta = \text{máx}\,(r_{LM_\theta} + p_{LM_\theta}, r_{PJ_\theta} + p_{PJ_\theta})$ donde LM_θ es la última operación planificada en la máquina requerida por θ;
3. Determinar la operación $\theta' \in A$ con el tiempo de finalización más temprano, si fuera planificada en el estado actual, es decir, $st_{\theta'} + p_{\theta'} \leq st_\theta + p_\theta, \forall \theta \in A$;
4. Sea m la máquina requerida por θ', y Cs el subconjunto de A cuyas operaciones requieren m;
5. Seleccionar $\theta'' \in Cs$ tal que $st_{\theta''} \leq st_\theta, \forall \theta \in Cs$;
6. Eliminar de Cs todas las operaciones θ tales que $st_\theta \geq st_{\theta''} + \delta((st_{\theta'} + p_{\theta'}) - st_{\theta''})$;
7. Elegir de alguna forma un $\theta^* \in Cs$;
8. Planificar θ^* lo más temprano posible para construir la planificación parcial que corresponde al siguiente estado, es decir, $r_{\theta^*} = st_{\theta^*}$;
9. Eliminar θ^* de A e insertar SJ_{θ^*} en A si θ^* no es la última operación de su trabajo;

fin mientras

devuelve La planificación construída;

Si $\delta = 1$ se tiene el algoritmo original y en consecuencia es posible generar cualquier planificación activa mediante la secuencia de selecciones no-deterministas adecuada. Sin embargo cuando $\delta < 1$ el algoritmo ya no genera soluciones en todo el espacio de búsqueda de las planificaciones activas y por tanto se puede perder la posibilidad de encontrar la solución óptima, y si $\delta = 0$ el algoritmo genera planificaciones densas. Esta idea se inspira

en una propuesta de Storer et al. en [136] para búsquedas en espacios de estados. En este caso el algoritmo busca en el espacio de las denominadas planificaciones activas parametrizadas.

La experiencia demuestra que en media las soluciones obtenidas de forma aleatoria son mejores a medida que se reduce el valor del parámetro de control. Sin embargo cuando lo utilizamos dentro de un algoritmo de búsqueda suele ser aconsejable no perder la posibilidad de encontrar la solución óptima, a no ser que necesitemos encontrar soluciones aceptables en un tiempo de ejecución extremadamente reducido.

La complejidad del algoritmo es del orden $O(N * M * \text{máx}(N, K))$, siendo K la complejidad de la operación de selección en el paso 7. En cada paso el algoritmo realiza una elección no determinista sobre el conjunto Cs de tareas críticas que pueden ser planificadas a continuación.

Como ya hemos comentado, un conjunto de planificaciones es dominante si contiene como mínimo una solución óptima. El conjunto de planificaciones que puede generar el algoritmo $G\&T$ para el JSP es dominante, ya que es capaz de generar cualquier planificación activa, y sabemos que el espacio de planificaciones activas contiene como mínimo una planificación óptima. Por el contrario, esto no ocurre utilizando el mismo algoritmo para el SDST-JSP, incluso aunque se utilice $\delta = 1$.

Entonces, con todo lo explicado hasta el momento, podemos representar la reducción del espacio de búsqueda en planificaciones activas para el problema JSP en un diagrama, como se muestra en la figura 3.1. Nótese que si se reduce excesivamente el valor de δ se pierde la posibilidad de encontrar una solución óptima.

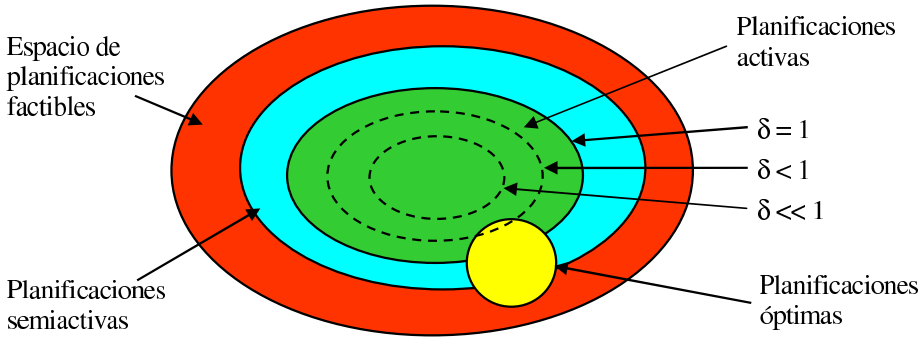


Figura 3.1: Esquema de los tipos de planificaciones para el JSP

3.3. Constructores de planificaciones para el SDST-JSP

El problema de aplicar el algoritmo $G\&T$ al SDST-JSP es que en este caso deja de ser capaz de generar todas las planificaciones activas. Esto es demostrado por Artigues y Lopez en [16], en donde además proponen dos extensiones al SDST-JSP del algoritmo $G\&T$ para el JSP. El conjunto de planificaciones generado por una de ellas no es dominante y el generado por la otra sí que lo es, pero el algoritmo es mucho más costoso computacionalmente. Por otra parte, en [17] Artigues et al. realizan un estudio detallado de todos los tipos de constructores de planificaciones para el SDST-JSP y estudian sus propiedades formales, entre otras si generan conjuntos de planificaciones dominantes o no.

3.3.1. Constructor de planificaciones activas EG&T para el SDST-JSP

El algoritmo 3.2 muestra uno de los algoritmos de construcción de planificaciones activas para el SDST-JSP. Es el constructor de planificaciones utilizado en el algoritmo genético híbrido propuesto por Vela et al. en [151]. Esta basado en el algoritmo $EG\&T$ propuesto por Artigues y Lopez en [16], y es una extensión simple y directa del algoritmo $G\&T$ para el JSP descrito por Giffler y Thompson en [55] pero que considera tiempos de setup. El conjunto de planificaciones que genera este algoritmo no es dominante. Nótese que en el caso del SDST-JSP no consideramos el parámetro de control δ.

En [17], Artigues et al. muestran varios ejemplos para demostrar que el conjunto de planificaciones que genera este algoritmo no sólo no es dominante, sino que además es posible que este algoritmo no genere una planificación activa, incluso aunque la instancia cumpla la desigualdad triangular de tiempos de setup. En ese mismo trabajo se muestra que para garantizar la obtención de una planificación activa con un constructor de planificaciones que vaya añadiendo operaciones al final, el conjunto conflicto no se puede restringir a las operaciones disponibles en un determinado instante de tiempo. Esto hace que el diseño de un constructor de planificaciones activas de este tipo sea bastante complejo.

Como ya hemos comentado, en [16] Artigues y Lopez proponen otras dos extensiones más del algoritmo $G\&T$ para el SDST-JSP, el conjunto de planificaciones generado por una de ellas tampoco es dominante y el generado por la otra sí. Sin embargo, las dos consumen más tiempo de ejecución que el algoritmo $EG\&T$, y por ello éste ha sido uno de los algoritmos elegidos en esta tesis para realizar el estudio experimental. De todas formas, el hecho de que

Algoritmo 3.2 Constructor de planificaciones *EG&T* para el SDST-JSP

Requiere: Un cromosoma C y una instancia de problema P

Produce: Una planificación construída a partir del cromosoma C para la instancia P

1. $A = \{\theta_{i1}, 1 \leq i \leq N\}$;

mientras $A \neq \emptyset$ **hacer**

2. $\forall \theta \in A$ sea $st_\theta = \text{máx}\left(r_{LM_\theta} + p_{LM_\theta} + S^m_{LM_\theta\theta}, r_{PJ_\theta} + p_{PJ_\theta}\right)$ donde LM_θ es la última operación planificada en la máquina requerida por θ;

3. Determinar la operación $\theta' \in A$ con el tiempo de finalización más temprano, si fuera planificada en el estado actual, es decir, $st_{\theta'} + p_{\theta'} \leq st_\theta + p_\theta, \forall \theta \in A$;

4. Sea m la máquina requerida por θ', y Cs el subconjunto de A cuyas operaciones requieren m;

5. Eliminar de Cs todas las operaciones θ tales que $st_\theta \geq st_{\theta'} + p_{\theta'} + S^m_{\theta'\theta}$;

6. Elegir de alguna forma un $\theta^* \in Cs$;

7. Planificar θ^* lo más temprano posible para construir la planificación parcial que corresponde al siguiente estado, es decir, $r_{\theta^*} = st_{\theta^*}$;

8. Eliminar θ^* de A e insertar SJ_{θ^*} en A si θ^* no es la última operación de su trabajo;

fin mientras

devuelve La planificación construída;

un constructor de planificaciones no sea capaz de generar por sí mismo la solución óptima no es un problema muy crítico cuando se utilizan estrategias de resolución metaheurísticas, pues éstas no suelen ser técnicas exactas.

3.3.2. Constructor de planificaciones activas SSGS para el SDST-JSP

El algoritmo *EG&T* es simple y proporciona planificaciones activas, pero el conjunto de planificaciones que genera no es dominante. También se utilizará en esta tesis un constructor de planificaciones diferente, el Serial Schedule Generation Scheme (SSGS) propuesto también por Artigues et al. en [17]. El SSGS itera sobre las operaciones en el orden que aparecen en un vector, y para cada una de esas operaciones elige el tiempo de inicio más temprano posible tal que satisfaga todas las restricciones con respecto a las operaciones planificadas anteriormente. La diferencia con un constructor de planificaciones semiactivas es que, en lugar de planificar las nuevas operaciones siempre al final, SSGS busca si la nueva operación

se puede introducir en algún "hueco" existente entre las tareas ya planificadas, y sólo si no existe ningún "hueco" lo suficientemente grande se planifica al final del todo, tal y como haría el constructor de planificaciones semiactivas.

El conjunto de planificaciones generado por el algoritmo SSGS sí que es dominante, tanto para el JSP como para el SDST-JSP. Es trivial demostrar esto, ya que el espacio de soluciones activas contiene como mínimo una solución óptima, y cualquier planificación activa se puede representar mediante un vector o una matriz, sin más que ordenar las tareas en forma de tiempo de inicio no decreciente.

El SSGS resulta ser mucho más eficiente en general que el $EG\&T$, y es el algoritmo utilizado por González et al. en el trabajo [62], en el cual se superan los resultados del trabajo de Vela et al. de [151], que utilizaba el constructor $EG\&T$.

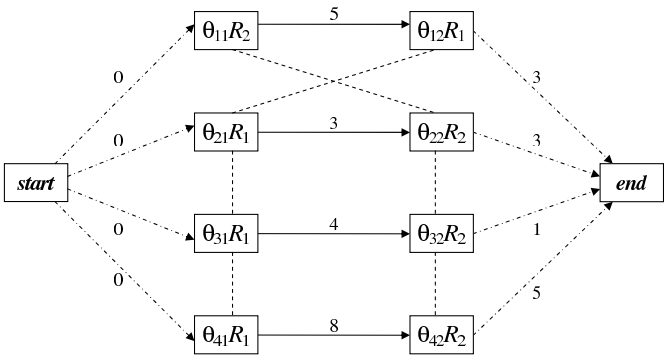
Por otra parte, el algoritmo SSGS proporciona planificaciones activas siempre y cuando la desigualdad triangular de tiempos de setup se verifique, como indican Artigues et al. en [17]. A continuación se muestran dos contraejemplos en los que dicha desigualdad no se verifica: en el primero de ellos veremos como se genera con SSGS una planificación que no es activa pero sí que es semiactiva, y en el segundo veremos como se genera una planificación que ni siquiera es semiactiva.

Contraejemplo 1: SSGS genera una planificación semiactiva pero no activa

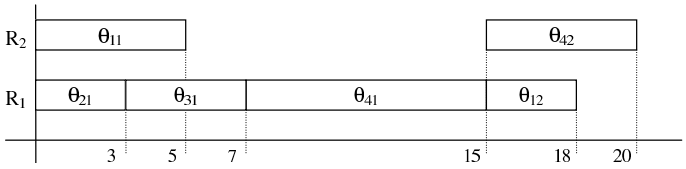
Vamos a considerar la instancia de SDST-JSP de la figura 3.2 (a). Es decir, una instancia con cuatro trabajos y dos recursos o máquinas. Las tareas θ_{12}, θ_{21}, θ_{31} y θ_{41} comparten el recurso 1 y dicho recurso no tiene tiempos de setup. Las tareas θ_{11}, θ_{22}, θ_{32} y θ_{42} comparten el recurso 2 y los tiempos de setup no nulos de dicho recurso son los siguientes: $S_{\theta_{11}\theta_{32}} = 8$, $S_{\theta_{32}\theta_{42}} = 2$, $S_{\theta_{11}\theta_{22}} = 2$, $S_{\theta_{22}\theta_{42}} = 2$. En cuanto a las duraciones de las tareas, $p_{\theta_{11}} = 5$, $p_{\theta_{12}} = 3$, $p_{\theta_{21}} = 3$, $p_{\theta_{22}} = 3$, $p_{\theta_{31}} = 4$, $p_{\theta_{32}} = 1$, $p_{\theta_{41}} = 8$, $p_{\theta_{42}} = 5$.

Supongamos que tenemos que planificar mediante SSGS el vector: $(\theta_{11}\theta_{21}\theta_{31}\theta_{41}\theta_{12}\theta_{42}\theta_{32}\theta_{22})$. Comenzamos planificando las seis primeras tareas. Ninguna de ellas es posible insertarla, por lo que todas ellas se van añadiendo a continuación de las tareas anteriormente planificadas. El resultado después de planificar las seis primeras tareas de ese vector se puede ver en el diagrama de Gantt de la figura 3.2 (b).

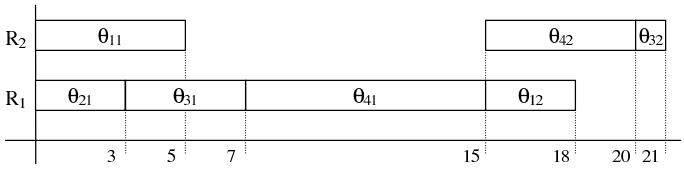
Ahora, cuando el método de inserción intenta planificar la tarea θ_{32}, no la puede insertar entre las tareas θ_{11} y θ_{42} por culpa de que $S_{\theta_{11}\theta_{32}} = 8$ es un tiempo de setup demasiado elevado y no cabe en ese hueco, por lo que se planifica al final, dando como resultado el



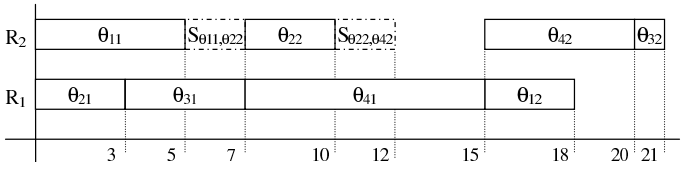
(a) Grafo disyuntivo de la instancia



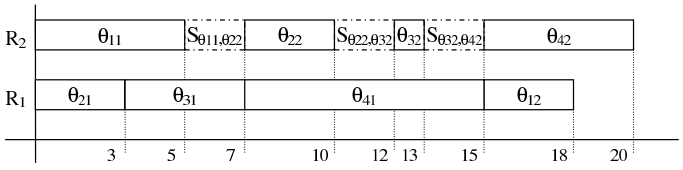
(b) Construcción de la planificación mediante SSGS. Paso 1



(c) Construcción de la planificación mediante SSGS. Paso 2



(d) Planificación final devuelta por SSGS (no es activa)



(e) Planificación activa a partir de (d)

Figura 3.2: Contraejemplo 1: SSGS genera una planificación semiactiva pero no activa

diagrama de Gantt de la figura 3.2 (c).

Y por último, se planifica la tarea θ_{22}, que se puede insertar entre las tareas θ_{11} y θ_{42}. Entonces, la planificación final devuelta por SSGS es la representada en la figura 3.2 (d). Sin embargo, esa planificación no es activa sino que es sólo semiactiva. No es activa porque la tarea θ_{32} ahora se podría poner entre las tareas θ_{22} y θ_{42} sin retrasar el tiempo de inicio de ninguna tarea, es decir de la forma presentada en la figura 3.2 (e).

La causa de que obtengamos una planificación que no es activa es que la instancia no cumple la desigualdad triangular de tiempos de setup, es decir que $S_{ij} \leq S_{ik} + S_{kj}$, siendo i, j y k tareas pertenecientes a la misma máquina, ya que observamos que $S_{\theta_{11}\theta_{32}} > S_{\theta_{11}\theta_{22}} + S_{\theta_{22}\theta_{32}}$.

Contraejemplo 2: SSGS genera una planificación que ni siquiera es semiactiva

En el contraejemplo anterior SSGS ha generado una planificación que no es activa, pero sí que era semiactiva. Sin embargo, también es posible generar una planificación que no sea ni activa ni semiactiva. En este caso supongamos que tenemos la misma instancia que en el anterior contraejemplo, pero esta vez queremos planificar el siguiente vector: $(\theta_{11}\theta_{21}\theta_{31}\theta_{41}\theta_{12}\theta_{32}\theta_{22}\theta_{42})$. Es decir, hemos cambiado el orden de las tres últimas tareas con respecto al vector del contraejemplo anterior. La planificación final de este vector dada por SSGS quedaría de la forma indicada en la figura 3.3.

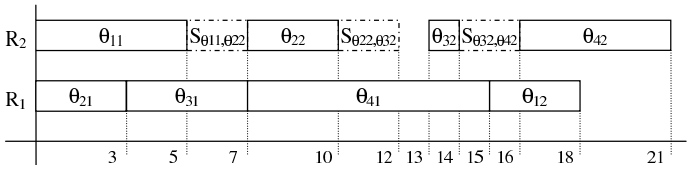


Figura 3.3: Contraejemplo 2: SSGS genera una planificación que ni siquiera es semiactiva

Es sencillo ver que en la planificación de la figura 3.3 se podría iniciar la tarea θ_{32} un instante de tiempo anterior, sin que sea necesario mover ninguna de las demás tareas ni cambiar el orden de ejecución de ninguna de ellas, por lo tanto esta planificación no es ni siquiera semiactiva. La causa de que se llegue a esta situación no deseable es que el tiempo de setup $S_{\theta_{11}\theta_{32}}$ es mayor que $S_{\theta_{11}\theta_{22}} + p_{\theta_{22}} + S_{\theta_{22}\theta_{32}}$, y al igual que en el caso anterior esto ocurre por culpa de no cumplirse la desigualdad triangular en los tiempos de setup.

Mecanismo de reparación cuando no se cumple la desigualdad triangular

Por lo que hemos visto hasta ahora, el SSGS puede construir planificaciones que no son ni siquiera semiactivas si la instancia no cumple la propiedad triangular en sus tiempos de setup. Esto puede suponer un problema ya que, por ejemplo, siempre existirá una planificación semiactiva estrictamente mejor, es decir, en la que todas las tareas comiencen en un instante de tiempo igual o menor. Además, la demostración de muchas de las propiedades de caminos críticos y bloques críticos presentadas en esta tesis presuponen que una tarea dentro de un bloque crítico terminará en el mismo instante de tiempo en que empieza la siguiente (lo cual ocurre siempre en una planificación semiactiva). Por estos y otros motivos debemos encontrar una solución a este problema. El método más simple e intuitivo es realizar una planificación semiactiva a partir de la ordenación proporcionada por SSGS. Esto nos aseguraría una planificación semiactiva, aunque no podríamos asegurar que sea activa. De todas formas éste será el método utilizado en esta tesis cuando tratemos con instancias que no cumplan la desigualdad triangular en sus tiempos de setup.

3.4. Conclusiones

En este capítulo hemos descrito las planificaciones semiactivas, activas y densas, y hemos presentado el algoritmo $G\&T$ propuesto por Giffler y Thompson en [55] para el JSP. El conjunto de planificaciones generado por este algoritmo es dominante para el JSP, ya que es capaz de generar todas las planificaciones activas posibles, y el conjunto de planificaciones activas incluye como mínimo una solución óptima. Sin embargo para el SDST-JSP el conjunto de planificaciones generado no es dominante, por lo que hemos propuesto varias posibles alternativas. El $EG\&T$ es una extensión directa del $G\&T$ y el conjunto de planificaciones que genera tampoco es dominante para el SDST-JSP. Por otra parte, el SSGS sí que genera un conjunto de planificaciones dominante tanto para el JSP como para el SDST-JSP. Sin embargo hemos visto que pueden surgir problemas al utilizar SSGS si la instancia del problema no cumple la propiedad de la desigualdad triangular en sus tiempos de setup.

Capítulo 4

MÉTODOS CLÁSICOS DE RESOLUCIÓN

4.1. Introducción

En este capítulo describiremos algunos de los métodos clásicos de resolución utilizados para resolver problemas de scheduling. Existen una gran cantidad de técnicas heurísticas aproximadas que, pese a que su proceso de búsqueda no garantiza la obtención de la solución óptima, ofrecen soluciones aceptables en tiempos de ejecución muy competitivos. En la primera parte de este capítulo se describirán dos de las más destacadas: las *Reglas de Prioridad* y el heurístico *Shifting Bottleneck*. Los algoritmos genéticos y las búsquedas locales también pertenecen al grupo de técnicas heurísticas, pero debido a que son los principales métodos de resolución utilizados en esta tesis, no se describirán aquí sino que se les dedicará su propio capítulo.

A continuación se detallarán algunas de las técnicas exactas utilizadas en la literatura. Estas técnicas, aunque aseguren obtener la solución óptima, tienen el grave inconveniente de que su tiempo de ejecución en instancias de tamaño medio o grande es prohibitivo. Se describirán técnicas de *Ramificación y Poda* y *Búsqueda Heurística en Espacios de Estados*, como el algoritmo A^*. Por último, se explicarán también las técnicas de *Programación con Restricciones*, y en concreto el funcionamiento de la aplicación *ILOG CPLEX CP Optimizer*,

una de las más conocidas y utilizadas en la actualidad.

4.2. Reglas de prioridad

Las *Reglas de Prioridad* permiten elegir el siguiente trabajo que será elegido en el proceso de construir una planificación. Estas reglas tienen la ventaja de que son muy rápidas, pero la desventaja de que la calidad de las soluciones obtenidas no suele ser muy buena. Sin embargo, debido a su fácil implementación y rapidez de ejecución, las reglas de prioridad son probablemente las estrategias mas aplicadas para resolver problemas de scheduling, sobre todo en combinación con otros métodos.

Una regla de prioridad asigna, como su propio nombre indica, prioridades a todos los trabajos que están esperando ser procesados en una máquina. Cada vez que una máquina queda libre, una regla selecciona la siguiente tarea que será planificada, es decir aquella tarea con mayor prioridad. Las reglas de prioridad también pueden ser empleadas para modificar el criterio de selección en distintos algoritmos a la hora de construir soluciones heurísticas. Por ejemplo podríamos emplearlas en el algoritmo $G\&T$ propuesto por Giffler y Thompson en [55] o en la construcción de una planificación con Prioridad de Jackson (*JPS*), como detalla Carlier en [35]. También es frecuente utilizarlas para generar las soluciones iniciales de otras metaheurísticas como pueden ser los algoritmos genéticos o la búsqueda local.

Las reglas de prioridad se pueden clasificar en reglas *estáticas* y reglas *dinámicas*. Las estáticas no dependen del tiempo sino sólo de los datos de los trabajos y las máquinas, mientras que las dinámicas sí que dependen del tiempo, por lo que en un instante de tiempo un trabajo j puede tener mayor prioridad que un trabajo k, y en otro instante de tiempo puede ocurrir lo contrario. También se pueden clasificar según la información que manejan, y según esta clasificación tenemos reglas *locales* y reglas *globales*. Las reglas locales sólo manejan información perteneciente a la cola de la máquina en la que el trabajo está esperando, mientras que las globales también manejan información relacionada con otras máquinas. Existe una gran cantidad de reglas de prioridad, ver por ejemplo el trabajo de Lawrence en [82] o el de Holthaus y Rajendram en [69]. Entre las reglas de prioridad básicas más utilizadas podemos destacar las siguientes, ordenadas alfabéticamente:

- *CP* (camino crítico): se utiliza cuando los trabajos están sujetos a restricciones de precedencia. Se selecciona el trabajo cuyo tiempo de procesamiento total es mayor, considerando únicamente las actividades precedentes a la actual.

- *CR* (ratio crítico): selecciona el trabajo con el menor ratio crítico, definido como el cociente entre el tiempo que falta hasta el due date del trabajo dividido por el tiempo de procesamiento restante del trabajo.

- *EDD* (instante de finalización más temprano): esta regla selecciona el trabajo que tenga el due date más próximo.

- *FCFS* (primera en llegar, primera en ser atendida): esta regla selecciona el trabajo que antes llega a la cola de la máquina. Ese será el trabajo con un tiempo de inicio más temprano, y por ello también es llamada a veces *ERD* (instante de inicio más temprano).

- *LFJ* (trabajo menos flexible): esta regla se suele utilizar cuando existen máquinas en paralelo que no son idénticas. Si cada trabajo j puede ser procesado por un subconjunto M_j de esas máquinas, se seleccionará aquel trabajo que pueda ser procesado por el menor subconjunto M_j. Es decir, el trabajo con menos alternativas de procesamiento posibles.

- *LNS* (mayor número de sucesores): esta regla se puede utilizar cuando los trabajos están sujetos a restricciones de precedencia, y selecciona el trabajo que tenga un mayor número de trabajos sucesores.

- *LPT* (tiempo de procesamiento más largo): esta regla da más prioridad a los trabajos cuyo tiempo de procesamiento es mayor. Cuando se tienen máquinas en paralelo esta regla tiende a equilibrar sus cargas de trabajo. Esto es debido a que suele resultar ventajoso mantener trabajos con tiempos de procesamiento cortos para más adelante, porque estos trabajos serán útiles al final para equilibrar la carga de las máquinas.

- *MS* (mínima holgura): selecciona el trabajo que en el momento de estar libre una máquina tiene el menor tiempo de holgura. El tiempo de holgura de un trabajo j se define como $max(d_j - pr_j - t, 0)$, siendo d_j su due date, pr_j su tiempo de procesamiento restante, y t el instante de tiempo actual. Es fácil comprobar que esta es una regla dinámica.

- *MWKR* (mayor trabajo restante): elige al trabajo con un mayor tiempo de procesamiento restante.

- *SIRO* (servicio en orden aleatorio): de acuerdo con esta regla, cuando una máquina esta libre el siguiente trabajo es seleccionado de forma aleatoria de entre los trabajos que están esperando para utilizar dicha máquina.
- *SPT* (tiempo de procesamiento más corto): se selecciona el trabajo cuyo tiempo de procesamiento es más pequeño.
- *SQNO* (cola más corta en la siguiente operación): esta regla se utiliza frecuentemente en el JSP, y selecciona el trabajo con la menor cola a la siguiente máquina en su ruta. La longitud de la cola a la siguiente máquina puede ser medida de distintas formas, por ejemplo el número de trabajos esperando en cola o la cantidad total de trabajo esperando en cola.
- *S/RPT* (holgura dividida entre el tiempo de procesamiento restante): la prioridad asignada a cada trabajo es la holgura, calculada de la misma forma que en la regla *MS*, pero esta vez dividida por el tiempo de procesamiento restante del trabajo.
- *SST* (tiempo de setup más corto): selecciona el trabajo que tendría un tiempo de setup más pequeño si se planificara a continuación.
- *SSTPT* (tiempo de setup y de procesamiento más corto): selecciona el trabajo que tenga más pequeña la suma de su tiempo de procesamiento restante y su tiempo de setup si se planificara a continuación.
- *TSPT* (tiempo de procesamiento más corto interrumpido): esta regla da prioridad a los trabajos de acuerdo con la regla *SPT*, con la excepción de los trabajos que han estado esperando más tiempo de un máximo fijado. Esos trabajos pasan al frente de la línea de espera en algún orden especificado (utilizando, por ejemplo, la regla *FCFS*).
- *WSPT* (tiempo de procesamiento ponderado más corto): los trabajos se seleccionan en orden decreciente de w_j/p_j, siendo w_j el peso asignado al trabajo j y p_j su tiempo de procesamiento en la máquina considerada. Cuando todos los trabajos tienen una ponderación igual, la regla *WSPT* es equivalente a la regla *SPT*.

Las reglas de prioridad básicas son útiles para encontrar buenas soluciones a problemas con un único objetivo, sin embargo para problemas con objetivos más complejos son necesarias reglas de prioridad compuestas. Estas reglas combinan varias reglas básicas asignándoles un peso a cada una de ellas. Un ejemplo lo encontramos en [152], en donde Vepsalainen

y Morton tratan de minimizar el weighted tardiness en problemas job-shop y utilizan dos reglas de prioridad. Una de ellas es la regla *ATC* (costo del retraso aparente), también detallada por Pinedo en [114], y que combina las reglas *WSPT* y *MS*. La otra es una versión con pesos de la regla *COVERT* (costo sobre el tiempo), regla inicialmente propuesta en [38] y que consiste en elegir el trabajo cuyo coste de retraso esperado por unidad de tiempo es mayor. Se comparan estas reglas en diversos experimentos con otras reglas como *FCFS*, *EDD*, *S/RPT* o *WSPT*, y los resultados muestran que la regla *ATC* mejora claramente a las demás.

Los primeros trabajos que abordaron la resolución del SDST-JSP utilizaban reglas de prioridad. Wilbrecht y Prescott en [157] proponen utilizar la regla de prioridad *SST* descrita anteriormente. Posteriormente, en [75], Kim y Bobrowski consideran varias medidas de rendimiento y proponen reglas de prioridad compuestas. En [104] Noivo y Ramalhinho-Lourenço también proponen nuevas reglas de prioridad y las comparan con otras anteriores. Otro ejemplo lo tenemos en [33], donde Brucker y Thiele proponen una regla de prioridad que es una extensión del algoritmo *G&T* de Giffler y Thompson propuesto en [55], y se utiliza dicha regla dentro de un método exacto.

Más recientemente en [17], Artigues et al. evalúan diferentes reglas de prioridad para el SDST-JSP utilizadas en combinación con varios Schedule Generation Schemas (SGS) de tipo semiactivo, activo y non-delay. Recordemos que los SGSs son algoritmos no-deterministas, y las reglas de prioridad se utilizan para hacer las selecciones no-deterministas cuando varias operaciones pueden ser elegidas para planificarse. Los resultados obtenidos en este artículo mejoran a los obtenidos por Brucker y Thiele en [33] y por Focacci et al. en [52].

4.3. Shifting Bottleneck

El heurístico Shifting Bottleneck fue propuesto por Adams et al. en [3], y ha demostrado ser muy efectivo en la resolución de problemas job shop. El método esta basado en la idea empírica de que el rendimiento de un sistema depende de la correcta utilización de los recursos que escasean. Básicamente el heurístico divide una instancia con M máquinas en M subproblemas OMS de una única máquina. Cada subproblema es capaz de capturar las características de la planificación de las restantes máquinas, gracias a que se construye a partir de la planificación original del JSP. Para obtener una instancia OMS se relajan en el problema original las restricciones de las máquinas cuyas tareas aún no han sido planificadas.

La idea general del método es resolver los subproblemas de forma individual y combinar las soluciones obtenidas para obtener la solución de la instancia original.

El heurístico planifica una máquina en cada iteración, eligiendo la más crítica (cuello de botella) de entre las que todavía no están planificadas. Para clasificar las máquinas se resuelve el problema de secuenciamiento de una máquina OMS asociado a cada una de ellas. La solución de dicho problema se puede calcular por ejemplo construyendo lo que se denomina una Planificación con Prioridad de Jackson (véase el trabajo de Carlier en [35]). Una vez resuelto el subproblema asociado a cada máquina, la que tenga la solución más costosa será la que se denomina cuello de botella. A continuación se fijan los arcos de la solución de ese subproblema en la instancia del JSP original, y además se reoptimizan el resto de máquinas que ya estuvieran planificadas en anteriores iteraciones, resolviendo de nuevo el subproblema OMS asociado a ellas pero teniendo en cuenta los nuevos arcos que hemos fijado. Esta reoptimización se repite hasta que no se produce ninguna mejora. Finalmente, el algoritmo termina cuando se han planificado todas las máquinas de la instancia.

Una de las ventajas del shifting bottleneck es que se puede adaptar fácilmente a problemas con diferentes características y funciones objetivo. En cuanto al estado del arte sobre este heurístico, podemos citar el trabajo [47], en el que Demirkol et al. estudian sus propiedades computacionales y lo aplican a diferentes funciones objetivo. Este heurístico también se ha extendido para que sea capaz de afrontar problemas job shop más cercanos a los entornos reales de producción. Por ejemplo, en [95] Mönch et al. tratan un problema job shop complejo con tiempos de setup y otras características como máquinas que procesan en paralelo y flujos de proceso reentrantes, y lo resuelven mediante un algoritmo genético hibridizado con un método de búsqueda local basado en el shifting bottleneck. Ives y Lambrecht en [72] también describen un método similar y consideran un número elevado de situaciones reales. En [133] Singer y Pinedo presentan un heurístico shifting bottleneck para minimizar el weighted tardiness en problemas job shop y obtienen resultados muy competitivos.

Este heurístico también se ha utilizado en la resolución del SDST-JSP. Por ejemplo, en [19] Balas et al. extienden al SDST-JSP el método shifting bottleneck propuesto por Adams et al. en [3] para el JSP clásico, y lo combinan con un procedemiento de búsqueda local guiada para mejorar las soluciones obtenidas. Más recientemente, en [20] Balas et al. utilizan un método muy parecido al propuesto por ellos mismos en [19] para tratar el problema SDST-JSP con release times y deadlines, y comparan sus resultados con los presentados por Ovacik y Uzsoy en [109] y en [110], obteniendo mejores resultados en varios

benchmarks de gran tamaño inspirados en la industria de los semiconductores. También se ha aplicado el heurístico shifting bottleneck en la minimización del weighted tardiness en el SDST-JSP, tal y como proponen Sun y Noble en [139].

En general el shifting bottleneck es un heurístico que ofrece muy buenos resultados, y por ello en esta misma tesis hemos utilizado varios de estos trabajos como referencia para evaluar nuestros algoritmos. En concreto, en el capítulo 8, dedicado a la minimización del makespan, se utilizarán resultados experimentales del trabajo [19]. También en el capítulo 9 se utilizarán como referencia los resultados del trabajo de [20] para evaluar nuestro algoritmo en la minimización del maximum lateness.

4.4. Métodos exactos

En la literatura se pueden encontrar diversos métodos para la resolución de problemas de optimización de forma exacta, por ejemplo los métodos de programación dinámica, los de búsqueda heurística en espacio de estados y los de ramificación y poda. En esta sección nos centraremos en estos dos últimos.

4.4.1. Ramificación y poda

Los algoritmos de ramificación y poda son posiblemente los métodos más empleados en la resolución exacta de problemas de optimización combinatoria. Tienen muchos elementos en común con otros algoritmos exactos como el A^*, pero tienen también algunas características propias. En un algoritmo de ramificación y poda cada estado se interpreta como un subconjunto de soluciones del problema original. En el estado inicial se tienen todas las soluciones y, mediante el proceso de ramificación, un conjunto de soluciones se descompone en la unión disjunta de varios conjuntos, hasta llegar a conjuntos unitarios que representan una única solución del problema. Este planteamiento hace que el espacio de búsqueda tenga estructura de árbol. En cambio, en el algoritmo A^* la búsqueda se plantea como un proceso en el que la solución se construye paso a paso, y por tanto el camino desde el estado inicial (que es el estado en el que aún no se ha hecho nada) hasta un estado intermedio representa parte de la solución del problema, y un estado intermedio se interpreta como un subproblema que se debe resolver para llegar a una solución del problema original.

Los métodos de ramificación y poda son capaces de explotar en cierta medida conocimiento del dominio del problema, mediante heurísticos y reglas de prioridad, con el objetivo

de guiar la búsqueda hacia regiones buenas del espacio de búsqueda. Los componentes fundamentales de los algoritmos de ramificación y poda son: un esquema de ramificación, un método de cálculo de cotas inferiores, un método de cálculo de cotas superiores, y una estrategia de control.

Mediante esta técnica se han podido resolver de forma satisfactoria un gran número de problemas de optimización complejos y problemas de satisfacción de restricciones. En la literatura existen multitud de propuestas y para resolver problemas de scheduling, y para el JSP tradicional vamos a destacar el algoritmo propuesto por Brucker et al. en [31] para la minimización del makespan. Otro ejemplo lo tenemos en [132], en donde Singer y Pinedo presentan y comparan varias técnicas de ramificación y poda para minimizar el total weighted tardiness. Tenemos un ejemplo más reciente en [25], en donde Beck propone un método llamado *Solution-Guided Multi-Point Constructive Search*, que es capaz de obtener soluciones de muy buena calidad. El método se inspira en la búsqueda tabú, y utiliza búsqueda con backtracking y reinicios, y para guiar la búsqueda utiliza heurísticos basados en texturas, además de las mejores soluciones encontradas hasta el momento en la búsqueda.

El algoritmo de ramificación y poda propuesto por Brucker et al. en [31] es uno de los mejores algoritmos exactos para la resolución del JSP con minimización del makespan. El algoritmo parte del grafo de restricciones del problema a resolver y va fijando arcos disyuntivos sucesivamente. Su principal característica es que emplea un esquema de ramificación que permite fijar arcos en la misma u opuesta dirección a la que tienen en el bloque crítico de una solución factible. Además, el algoritmo de Brucker explota la propagación de restricciones denominada *Selección Inmediata* (véase el trabajo de Carlier y Pinson en [36]). Este método permite fijar arcos disyuntivos adicionales con lo que se logra una drástica reducción del árbol de búsqueda. Por otra parte, el algoritmo utiliza métodos muy eficientes para la obtención de buenas cotas inferiores y superiores. El cálculo de las cotas inferiores se basa en el problema de secuenciamiento de una máquina relajado. La solución óptima de este problema se obtiene mediante la planificación de Jackson (Jackson's Preemptive Schedule, *JPS*) obtenida en tiempo polinomial por el algoritmo propuesto por Carlier en [35]. Las cotas superiores se obtienen empleando el algoritmo voraz $G\&T$ propuesto por Giffler y Thompson en [55]. El algoritmo de Brucker resuelve fácilmente casi todas las instancias de tamaño 10×10 así como algunas instancias mayores. Sin embargo, las ideas de este algoritmo no son fácilmente generalizables cuando la función objetivo es diferente del makespan. Como curiosidad, el conjunto de instancias seleccionadas por Applegate y Cook en

[9] son consideradas muy difíciles debido a que no son resueltas por este algoritmo, con la única excepción de la famosa instancia $FT10$, que es famosa porque fue propuesta en 1963 por Fisher y Thompson en [51], pero la solución óptima no fue demostrada hasta 1989 por Carlier y Pinson en [36].

Recientemente también se han propuesto algoritmos híbridos que combinan búsqueda exacta y búsqueda local. Por ejemplo, Streeter y Smith en [137] combinan una búsqueda local con el algoritmo de ramificación y poda propuesto por Brucker et al. en [31], y son capaces de obtener rápidamente soluciones muy buenas para instancias de tamaño pequeño y mediano. Beck et al. en [26] proponen combinar el método de búsqueda propuesto por Beck en [25] con búsqueda tabú, y en dicho trabajo se puede observar que la combinación de los dos métodos produce mejores resultados que cada uno de ellos por separado.

En cuanto al SDST-JSP, por el momento los métodos exactos más relevantes son los algoritmos de ramificación y poda de Brucker y Thiele en [33], de Focacci et al. en [52] y de Artigues et al. en [13], [14] y [15].

Brucker y Thiele proponen en [33] un método que extiende el algoritmo de Brucker et al. de [31] del JSP clásico a un GSSP (General Shop Scheduling Problem) con tiempos de setup dependientes de la secuencia, que incluye al SDST-JSP como caso particular. Proponen un conjunto de 15 instancias que es el benchmark más utilizado desde entonces. Este conjunto esta basado en las instancias LA01-15 propuestas para el JSP clásico por Lawrence en [82]. Las cotas inferiores se calculan resolviendo una relajación del problema mediante la planificación de Jackson (Jackson's Preemptive Schedule, *JPS*), tal como proponen Carlier y Pinson en [36]. La ramificación se basa en invertir operaciones en el camino crítico de una planificación factible a partir del nodo actual, y las planificaciones factibles se calculan con una regla de prioridad que es una extensión del algoritmo $G\&T$ de Giffler y Thompson ([55]).

En [52] Focacci et al. proponen un método para una variante del problema que considera recursos alternativos para las operaciones y tratan de minimizar el tiempo total de setup y el makespan.

El método más reciente y eficiente lo proponen Artigues y Feillet en [15]. Este algoritmo se basa en la cooperación entre técnicas de propagación de restricciones y un algoritmo exacto truncado que resuelve relajaciones del TSPTW (Travelling Salesman Problem with Time Windows). Para obtener planificaciones factibles utilizan una regla de prioridad basada en las soluciones del TSPTW. El esquema de ramificación considera el conjunto de operaciones

que crean un conflicto con la operación no planificada con el menor tiempo de inicio. El algoritmo es capaz de resolver de forma óptima ocho instancias del conjunto BT, mientras que los métodos de Focacci et al. de [52] o de Brucker y Thiele de [33] sólo son capaces de resolver las cinco instancias más pequeñas. Además, el algoritmo mejora la mejor cota inferior conocida para ocho instancias, y mejora la mejor cota superior conocida para otras tres. Debido a su gran eficiencia, es uno de los métodos con los que nos compararemos en el capítulo 8.

4.4.2. Búsqueda heurística en espacios de estados

La búsqueda heurística en espacios de estados es una de las técnicas clásicas de la Inteligencia Artificial. Es una técnica que permite resolver problemas de optimización combinatoria incorporando conocimiento específico del dominio del problema. A la hora de diseñar un sistema de búsqueda en espacios de estados para resolver un determinado problema debemos considerar los siguientes componentes: los estados, las reglas u operadores y la estrategia de control. Los estados representan las sucesivas situaciones o problemas residuales por los que se pasa durante la resolución de un problema. Las reglas u operadores permiten moverse de un estado a otro, es decir obtener los sucesores de un estado. El coste asociado a cada regla y operador depende de la función objetivo del problema a resolver.

El conjunto de estados y operadores definen el espacio de búsqueda que, en el paradigma de búsqueda en espacios de estados, tiene forma de grafo dirigido simple. En problemas reales, este grafo es tan grande que no es posible representarlo en la memoria del ordenador, por ello se representa de forma implícita a partir de un único estado inicial y del conjunto de operadores. En general, hay varios nodos del espacio de búsqueda que representan las soluciones del problema y estos son los objetivos. En la práctica tampoco es posible almacenarlos de forma explícita ya que normalmente o no se conocen o son muchos, por lo que es preciso disponer de una función que permita caracterizarlos, es decir, una función que compruebe si un determinado estado es objetivo.

La estrategia de control es la que decide el orden de exploración de los estados. Podemos emplear una estrategia de control inteligente o informada, que mediante heurísticos nos debería llevar a explorar primero los nodos que se encuentran en el camino que nos lleva a una solución óptima. Por el contrario una búsqueda no informada o ciega (por ejemplo los esquemas clásicos de búsqueda primero en anchura y primero en profundidad), considerará igual de buenos todos los sucesores de cada estado, y por ello normalmente necesitará explo-

rar o expandir un número de estados mucho mayor que la búsqueda informada para llegar a una solución. El objetivo de los algoritmos de búsqueda inteligente es encontrar buenas soluciones, y si es posible la solución óptima, explorando el menor número posible de nodos del grafo. Los heurísticos nos pueden ayudar por ejemplo para decidir el nodo más prometedor, de entre todos los nodos candidatos a ser expandidos. Sin embargo si un heurístico es demasiado costoso computacionalmente, puede no merecer la pena incluso aunque sea capaz de tomar muy buenas decisiones. Lo más habitual para diseñar un buen heurístico es relajar algunas restricciones del problema original hasta tener una versión del problema lo suficientemente simple como para poder ser resuelta con un algoritmo polinomial. También hay que tener en cuenta que, cuanto menor sea la relajación realizada, más se parecerá el problema relajado al problema original, y por tanto más preciso será el heurístico.

El algoritmo de búsqueda primero el mejor, o BF (Best First), es una especialización del algoritmo general de búsqueda en grafos. Se parte de un nodo inicial y se dispone de un conjunto finito de operadores capaces de producir los sucesores de un nodo. El algoritmo utiliza estructuras para almacenar el grafo desarrollado hasta el momento, y el mejor camino desde cada nodo al nodo inicial. Los nodos candidatos a ser expandidos se ordenan mediante una función de evaluación heurística f, que tiene el cometido de ordenar los nodos según lo prometedores que sean. El nodo más prometedor será el sigueinte que se visitará.

El algoritmo A^* es una especialización del algoritmo general de búsqueda en grafos en su versión BF (Best First). Nilsson en [103] o Pearl en [111] ofrecen una descripción detallada del método. En el algoritmo A^*, la función de evaluación $f(n)$ es una estimación del coste del camino solución condicionado a pasar por el nodo n que se esta evaluando. Para ello descompone la función f en $f(n) = g(n) + h(n)$, siendo $g(n)$ el coste del camino más corto desde el inicial a n encontrado hasta el momento en la búsqueda, y $h(n)$ una estimación positiva del coste del camino más corto desde n al objetivo más cercano a n, tal que $h(n) = 0$ si n es un objetivo. A la función h se le suele denominar función heurística, y la clave del buen funcionamiento de un algoritmo A^* es que dicha función sea lo más precisa posible y que no consuma mucho tiempo de ejecución.

Para que el algoritmo A^* sea admisible, es decir que devuelva la solución óptima (con un camino desde el nodo inicial con mínimo coste posible) siempre que dicha solución exista, se debe emplear una función h admisible. Una función heurística h se denomina admisible si y sólo si $h(n) \leq h^*(n) \forall n$, siendo $h^*(n)$ la función que determina el coste exacto desde el nodo n hasta el objetivo más cercano a n. Por ejemplo, las estimaciones heurísticas obtenidas

mediante relajaciones del problema original son admisibles.

Debido a la complejidad de muchos problemas no es posible diseñar buenos heurísticos, es decir que sean muy precisos y consuman poco tiempo de ejecución. Por este motivo suele ser necesario expandir una gran cantidad de nodos para alcanzar una solución, lo cual implica un elevado tiempo de ejecución y un alto requerimiento de espacio de almacenamiento. De hecho, en problemas de cierta complejidad se suelen agotar los recursos computacionales sin que el algoritmo encuentre una solución. Para resolver este problema se han propuesto diversas alternativas que renuncian a la admisibilidad a cambio de lograr una mayor eficiencia del algoritmo A^*. Tienen la ventaja de que son capaces de obtener buenas soluciones en problemas muy complejos, pero la desventaja de que en dichos problemas no podrán garantizar la optimalidad de la solución que ofrecen. Un ejemplo de estas técnicas es el algoritmo $A\varepsilon^*$.

Por otra parte, incluso aunque empleemos heurísticos muy precisos, la memoria consumida por el algoritmo crece exponencialmente con la profundidad, y éste es el mayor problema del algoritmo A^*. También se han propuesto diferentes técnicas para limitar la cantidad de memoria consumida, como por ejemplo el algoritmo IDA^* (Iterative Sampling A^*), el algoritmo SMA^* (Simplified Memory-Bounded A^*), o el método de búsqueda en frontera.

En los trabajos [125] y [126], Sierra et al. presentan un algoritmo A^* que utiliza técnicas de poda basadas en relaciones de dominancia entre estados, y lo aplican a la minimización del makespan y del total flow time en el problema job-shop clásico. En la minimización del total flow time consiguen resolver de forma exacta menos instancias que las que resuelven en el caso del makespan, debido a que es una función objetivo más compleja, sin embargo en la minimización del total flow time el algoritmo propuesto es muy competitivo y mejora a otros métodos de la literatura. A continuación describiremos este método, ya que es uno de los algoritmos con los que nos compararemos en el estudio experimental realizado en el capítulo 11.

En resumen, es un algoritmo de búsqueda exacta best-first, que utiliza una estimación heurística admisible obtenida mediante relajaciones a problemas con una única máquina (OMS). Se realiza una relajación para cada máquina m y después se calcula el heurístico de esta forma:

$$h(n) = \max_{m \in R} \triangle_m; \tag{4.1}$$

donde $\triangle_m$ denota el coste óptimo de la instancia del problema OMS asociado a la

máquina m. Este valor se obtiene en tiempo polinomial mediante el algoritmo propuesto por Carlier y Pinson en [36] y en [37]. El algoritmo A^* se combina con un método para podar nodos basado en relaciones de dominancia entre estados del árbol de búsqueda. Dados dos estados n_1 y n_2, decimos que n_1 domina a n_2 si y sólo si la mejor solución que se puede alcanzar desde el estado n_1 es mejor, o al menos igual, que la mejor solución que se puede alcanzar desde n_2. En algunas ocasiones es posible detectar esta situación, lo que permitiría realizar la poda temprana del nodo dominado. Sierra et al. detallan las condiciones que se deben verificar para que un nodo domine a otro, y experimentalmente comprueban que dichas técnicas de poda permiten reducir considerablemente tanto el espacio de búsqueda efectivo como el tiempo de ejecución. El algoritmo resultante es denominado $A^* - PD$, y es capaz de resolver de forma óptima instancias de tamaños hasta 10×5 y 9×9. Para instancias más grandes, la memoria se suele agotar antes de alcanzar la solución óptima.

Para poder dar soluciones en instancias de mayor tamaño, en [125] y [126] Sierra et al. proponen una variante no admisible del algoritmo. Dicha variante se basa en una estrategia de ponderación heurística, que depende de cada problema en particular y consiste en ponderar todos los términos $\triangle_m$ de la expresión 4.1, en lugar de elegir simplemente el máximo. Si consideramos que estos valores se ordenan de mayor a menor como $\triangle_1 \geq \cdots \geq \triangle_M$, entonces la función de ponderación heurística se calcula como:

$$h_{wi}(n) = \triangle_1 + \sum_{2 \leq i \leq M} \frac{\triangle_i}{2^{w_i + \delta}}, w_i > 0. \tag{4.2}$$

donde w_i y δ son parámetros. Es sencillo comprobar que $h_{wi}(n) \geq h(n)$. En [125] Sierra llama a este método ponderación disyuntiva, y el algoritmo resultante es denominado $A^* - DW$. Parece razonable elegir los valores de los parámetros w_i de tal forma que $\triangle_2 \ldots \triangle_M$ contribuyan menos que $\triangle_1$ a la estimación ponderada, y por ello en ese trabajo proponen inicializar los parámetros de la siguiente forma: $w_i = (i - 1)$, $2 \leq i \leq M$ y $\delta = 0$. El método utilizado en su estudio experimental es iterar sobre δ a intervalos de 0,2 hasta que o bien $\delta = 2$ o la memoria se agote. En cada una de las iteraciones, el algoritmo finaliza cuando se alcanza la primera solución, que en general no será la óptima. Finalmente, se calculan tantas soluciones como se puede con el valor más grande de δ que ha sido capaz de resolver el problema sin agotar la memoria.

4.4.3. Programacion con restricciones

La programación con restricciones es un método ampliamente utilizado para la descripción y resolución de problemas combinatorios particularmente difíciles, por ejemplo en las áreas de planificación y scheduling. Es un paradigma de programación que se basa en la especificación de un conjunto de variables, y otro conjunto de restricciones que deben ser satisfechas por cualquier solución del problema. Mediante este paradigma se han desarrollado técnicas exactas muy efectivas para resolver el JSP. Estos métodos combinan búsqueda con mecanismos de propagación de restricciones, que son capaces de reducir en gran medida el espacio de búsqueda.

Algunos ejemplos de métodos de programación de restricciones para el JSP son los propuestos por Dorndorf et al. en [48], por Laborie en [80] o por Baptiste et al. en [23]. En cuanto a las estrategias de búsqueda, el backtracking ha sido muy utilizado en este dominio, y se han propuesto un cierto número de heurísticos que ordenan variables (para determinar el proceso de ramificación) o que ordenan valores (para guiar la búsqueda), por ejemplo el heurístico basado en texturas propuesto por Beck y Fox en [24] o el heurístico basado en holguras propuesto por Smith y Cheng en [134]. En estos ejemplos, la mayoría del tiempo de computación se dedica a ordenar variables y se utilizan reglas de prioridad simples para guiar la búsqueda.

ILOG CPLEX CP Optimizer

El IBM ILOG CPLEX Optimization Studio es un paquete de optimización muy utilizado en estos últimos años. En particular, el ILOG CPLEX CP Optimizer es una librería incluída en ese paquete y esta dedicada a la programación con restricciones. Incluye una serie de clases y métodos para definir y resolver problemas de satisfacción de restricciones, y de optimización y scheduling.

Para encontrar una solución a un problema utilizando CP Optimizer, los dos primeros pasos son describir y modelar el problema mediante variables y restricciones. Cada variable representa la información desconocida de un problema y tiene un dominio de posibles valores que puede tomar. Las restricciones son los límites para las combinaciones de valores entre las diferentes variables. También se debe definir un objetivo para el problema.

Una vez modelado el problema se puede proceder a su resolución, es decir a encontrar la mejor solución posible o la solución óptima. Encontrar una solución significa asignar un valor para cada una de las variables, de tal forma que cada valor se encuentre en el dominio

de la variable, que todas las restricciones se satisfagan, y que se maximice o minimice el objetivo definido. El espacio de búsqueda esta formado por todas las posibles combinaciones de valores para las variables. El CP Optimizer utiliza dos técnicas para resolver problemas de optimización: búsqueda constructiva y propagación de restricciones. Utiliza además dos tipos de propagación de restricciones: una inicial y otra durante el proceso de búsqueda.

El término propagación de restricciones se refiere a dos procesos diferenciados: por una parte la reducción del dominio de algunas de las variables, y por otra parte comunicar dichas reducciones a las restricciones. Al comenzar su ejecución, CP Optimizer realiza una propagación de restricciones inicial, con el objetivo de eliminar valores del dominio que es imposible que vayan a ser utilizados en ninguna solución. Tras la propagación de restricciones inicial, el espacio de búsqueda se reduce.

A continuación, CP Optimizer busca una solución que se encuentre en la porción restante del espacio de búsqueda. La idea es recorrer un árbol de búsqueda, en el que la raíz es el punto de partida, cada rama representa una alternativa en la búsqueda, y cada hoja es una combinación de valores para las variables. El CP Optimizer utiliza una estrategia de búsqueda constructiva, cuya idea es ir asignando valores variable por variable y comprobar si puede llegar a una solución a través de esas asignaciones o ramas del árbol. Si no es posible llegar a una solución, se intentan diferentes valores, es decir, se busca en otras ramas del árbol.

Durante la búsqueda también se realiza una propagación de restricciones, aunque es diferente a la realizada inicialmente. En este caso se eliminan todos los valores de los actuales dominios que violan las restricciones, pero teniendo en cuenta las variables cuyo valor ya hemos asignado en la rama actual del arbol de búsqueda. Es decir, en este caso cuando se eliminan valores de los dominios, únicamente se eliminan en la actual rama del árbol, y si alguna variable se queda sin posibles valores significa que a través de esta rama no se puede encontrar una solución.

CP Optimizer implementa tres estrategias de búsqueda diferentes. Por defecto utiliza la denominada “Restart”, que consiste en reiniciar la búsqueda constructiva cada cierto tiempo, para guiarla hacia la solución óptima. También se puede utilizar la estrategia “Depth-First”, que explora de forma exhaustiva cada rama del árbol de búsqueda hasta que encuentra una solución, o bien se prueba que no existe una solución en ese subárbol. Esta estrategia es en general menos eficiente, debido a que es muy costoso escapar de una rama del árbol en la que no existen buenas soluciones, si el subárbol definido por dicha rama es muy grande.

Una última opción es la estrategia "Multi-Point", que crea un conjunto de soluciones, y posteriormente combina las soluciones del conjunto para ir obteniendo mejores soluciones. Este tipo de búsqueda obtiene soluciones más diversas, aunque no necesariamente prueba la optimalidad de las soluciones encontradas.

En esta tesis hemos considerado el CP Optimizer para obtener resultados con los que poder comparar nuestros algoritmos en benchmarks o en problemas para los que no nos podamos comparar con otros métodos de la literatura. En particular, lo utilizaremos en la sección 8.4.3, en la sección 10.4 y en la sección 11.4, para la minimización del makespan, weighted tardiness y total flow time, respectivamente. Hemos optado siempre por la estrategia de búsqueda por defecto, "Restart", ya que experimentalmente los resultados han sido bastante mejores con ella. Además, hemos utilizado siempre el valor "Extended" para el parámetro "NoOverlapInferenceLevel". Este parámetro controla el nivel de reducción de dominio aplicado en las restricciones utilizadas para modelar el SDST-JSP. El valor "Extended" implica que se consumirá un mayor tiempo de ejecución pero la reducción del dominio será mayor, y lo hemos configurado de esa forma porque experimentalmente los resultados han sido ligeramente mejores.

Este método es capaz de encontrar la solución óptima y probar su optimalidad, siempre que el tiempo de ejecución sea lo suficientemente elevado. Sin embargo, el tiempo de ejecución necesario en problemas de tamaño mediano o grande podría ser enorme, y por lo tanto en los experimentos realizados en esta tesis las soluciones ofrecidas por este método serán subóptimas.

4.5. Conclusiones

En esta sección hemos resumido algunos de los métodos clásicos de resolución frecuentemente empleados en problemas de optimización. Las reglas de prioridad son un método ampliamente utilizado debido a su sencillez. Se basan en construir una planificación trabajo por trabajo, y en cada paso se ordenan los trabajos candidatos según la regla utilizada. Aunque por sí solas las reglas de prioridad no suelen obtener muy buenos resultados, es muy habitual utilizarlas en combinación con otras técnicas.

El heurístico shifting bottleneck consiste en planificar cada máquina de forma separada, y para cada una de ellas se resuelve de forma óptima una relajación del problema original. Cada vez que se optimiza una nueva máquina se vuelven a reoptimizar el resto de máquinas,

y el proceso continúa hasta que no se encuentre mejora. Esta técnica ha obtenido muy buenos resultados en una gran variedad de problemas.

También describimos métodos que proporcionan soluciones exactas, como por ejemplo métodos ramificación y poda y el algoritmo A^*. Estas técnicas tienen la ventaja de que garantizan encontrar la solución óptima si se dispone de una cantidad de recursos (tiempo computacional y memoria) lo suficientemente grande, pero tienen el inconveniente de que en problemas de un cierto tamaño no suelen obtener la solución óptima ya que el espacio de búsqueda es extremadamente grande, y por tanto la cantidad de recursos necesaria también. Sin embargo estos métodos exactos se han utilizado de forma satisfactoria en una gran cantidad de problemas.

Por último, hemos mostrado un breve resumen sobre las técnicas de programación con restricciones y describimos el funcionamiento de la aplicación ILOG CPLEX CP Optimizer, ya que es uno de los métodos de resolución con los que nos compararemos en algunos de los estudios experimentales realizados en esta tesis.

Capítulo 5

EL ALGORITMO GENÉTICO

5.1. Introducción

El JSP es un ejemplo de problema de optimización combinatoria, y se ha intentado resolver mediante diversas técnicas heurísticas. En particular, los Algoritmos Genéticos (AGs) son un método muy utilizado debido a que pueden tratar espacios de búsqueda muy grandes y también porque se pueden combinar fácilmente con otras técnicas, como la búsqueda local. Además, los algoritmos genéticos permiten aprovechar cualquier tipo de conocimiento heurístico del dominio del problema. De esta forma, los AGs son competitivos con los métodos más eficientes para resolver el JSP, como podemos ver por ejemplo en los trabajos de Bierwirth en [27], de Mattfeld en [88], de Yamada y Nakano en [160], de Aydin y Fogarty en [18], de Varela et al. en [150] o de González et al. en [63].

Los algoritmos genéticos fueron introducidos por Holland y algunos de sus colaboradores de la Universidad de Michigan en los años 70, y están inspirados en la evolución de los seres vivos. Son un caso particular de algoritmo evolutivo, y constituyen una estrategia de búsqueda de propósito general que permite resolver problemas de naturaleza combinatoria, como los problemas de Job-Shop Scheduling. En esencia, utilizan una estrategia de búsqueda estocástica en un espacio de soluciones potenciales de un problema que trata de modelar las leyes de la evolución natural, en particular la herencia genética y la adaptación al entorno.

Un algoritmo genético realiza una búsqueda multidireccional manteniendo una población de soluciones potenciales y favoreciendo el intercambio de información entre las distintas

direcciones de búsqueda. La población sufre una evolución simulada: en cada generación los individuos buenos se reproducen mientras que los malos se mueren; en este proceso se generan individuos nuevos que heredan algunas características de sus progenitores, mientras que otras características son propias de cada individuo.

En principio un individuo se representa mediante un vector de símbolos, aunque en el caso más simple se utiliza una representación binaria, es decir, un vector de ceros y unos. Siguiendo el lenguaje de la biología, a la representación de un individuo se le suele llamar cromosoma y a cada uno de los símbolos gen.

Para discriminar los individuos buenos de los malos se utiliza la función fitness o de evaluación que juega el papel del entorno. Esta función otorga un valor numérico a cada individuo que emula su adaptación al entorno. Para ello, primero debe traducir la representación interna del individuo (genotipo) a la representación externa (fenotipo), es decir, la potencial solución del problema.

Para poder resolver un problema mediante un algoritmo genético, se deben definir un cierto número de componentes, entre otros:

- Una representación genética de las soluciones potenciales del problema.
- Un modo de crear la población inicial.
- Una función de evaluación que juega el papel del entorno, permitiendo clasificar a los individuos en términos de su fitness.
- Operadores genéticos de selección, cruce y mutación que simulan la evolución de las poblaciones.
- Valores de los parámetros del algoritmo: tamaño de la población, número de generaciones, probabilidad de cruce y de mutación, etc.

La gran utilidad que tienen los algoritmos genéticos como estrategias de resolución de problemas se basa en los excelentes resultados experimentales que producen. Otra de sus grandes ventajas es que su diseño permite explotar las técnicas de procesamiento en parelelo.

No obstante, existen algunos resultados teóricos que avalan su eficiencia. En el caso del algoritmo genético simple con representación binaria, el *Teorema de los Esquemas* (ver el trabajo de Holland en [68]) garantiza de una forma estadística que para valores razonables de los parámetros del algoritmo el fitness medio de la población mejora en las sucesivas

generaciones. Un esquema identifica a un subconjunto de cadenas que son idénticas en determinadas posiciones, por ejemplo el esquema 1*10*1** describe el conjunto de todas las cadenas de longitud 8 que tienen un 1 en la primera, tercera y sexta posiciónes, y un 0 en la cuarta posición. El teorema de los esquemas demuestra que la presencia de los esquemas cortos (es decir poca distancia entre el primer y último elementos específicos) y de bajo orden (es decir pocos elementos específicos) que tengan un fitness por encima de la media se incrementará exponencialmente en sucesivas generaciones del algoritmo genético, mientras que la presencia de los esquemas con un fitness por debajo de la media decrece exponencialmente.

5.2. Estructura general de un algoritmo genético

El algoritmo 5.1 muestra la estructura del genético que utilizaremos en esta tesis. Esta estructura es similar al algoritmo genético básico que se describe en la literatura; véase por ejemplo los algoritmos de Holland en [68], de Goldberg en [60] o de Michalewicz en [93]. En líneas generales, la estrategia operativa consiste en partir de una población inicial de individuos, cada uno de los cuales representa una posible solución del problema. Estos individuos se evalúan mediante una función (fitness) que indica la calidad de la solución o grado de adaptación del individuo al entorno. A partir de esta situación inicial, se realizan una serie de iteraciones o generaciones. En cada iteración se construye una nueva generación de individuos a partir de la generación anterior, aplicando los operadores genéticos de selección, recombinación y aceptación. Estos operadores se pueden implementar de muchas formas y en principio son independientes unos de otros, pero en la práctica todos ellos se deben elegir pensando en el efecto que puedan tener en los demás, para así conseguir un buen resultado global. Al final de la ejecución se llegará a una población que, si el algoritmo converge adecuadamente, estará compuesta en su mayor parte por muy buenos individuos. La solución que se devuelve será el mejor individuo evaluado a lo largo de la ejecución.

Un algoritmo genético convencional como el descrito en esta sección suele obtener resultados moderadamente buenos. Pero se pueden obtener mejoras significativas hibridizándolo con otros métodos, como por ejemplo la búsqueda local. En ese caso, el algoritmo genético híbrido se denomina también algoritmo memético. Esta hibridación se consigue aplicando el algoritmo de búsqueda local a todos los cromosomas justo después de que cada uno se genere. Es decir, en el punto 5 del algoritmo 5.1 un algoritmo genético convencional utilizaría sobre el cromosoma un constructor de planificaciones estándar y calcularía la función objetivo de

Algoritmo 5.1 Algoritmo genético

Requiere: Una instancia de un problema de scheduling P

Produce: Una planificación H para la instancia P

1. Generar la población inicial;
2. Evaluar la población;

mientras No se cumpla el criterio de parada **hacer**

3. Elegir cromosomas de la población actual;
4. Aplicar el operador de recombinación a los cromosomas elegidos en el paso 3 para generar otros nuevos;
5. Evaluar los cromosomas generados en el paso 4;
6. Aplicar el criterio de aceptación al conjunto de cromosomas elegidos en el paso 3 junto con los cromosomas generados en el paso 4;

fin mientras

devuelve La planificación del mejor cromosoma evaluado hasta el momento;

la planificación, sin embargo un algoritmo memético aplicará el método de búsqueda local a dicha planificación y calculará la función objetivo del individuo mejorado.

5.3. Revisión bibliográfica sobre algoritmos genéticos en problemas de scheduling

Los AGs son algoritmos muy versátiles, relativamente sencillos de programar, y que producen buenos resultados en general. Por todos estos motivos se han aplicado en numerosos trabajos sobre problemas de scheduling. A continuación ofrecemos una breve revisión bibliográfica sobre aplicaciones recientes. Primero nos centraremos en algunos trabajos sobre el job shop clásico, que es uno de los problemas de scheduling más tratados en la literatura, y después consideraremos algunas extensiones del job shop, entre otras el SDST-JSP.

En [116], Ponnambalam et al. presentan un método para estimar los parámetros de un algoritmo genético para el JSP. Wang y Zheng utilizan en [153] un operador de cruce muy efectivo para una representación basada en operaciones. El clásico operador de mutación es reemplazado por el algoritmo de Metropolis, en el que se basa el enfriamiento simulado, y esto ofrece una cierta capacidad probabilística de ampliar la vecindad utilizada en la búsqueda. Murovec y Suhel en [97] también tratan la minimización del makespan en el

job shop tradicional mediante una combinación de algoritmo genético y búsqueda local. En este caso la novedad es que la estructura de vecindad propuesta, en lugar de rechazar los vecinos que dan lugar a soluciones no factibles, aplican un mecanismo de reparación a dichas soluciones para convertirlas en factibles. En [84], Liu et al. proponen un genético mejorado para resolver el JSP, llamado el algoritmo genético híbrido Taguchi. Amirthagadeswaran y Arunachalam proponen en [7] un nuevo método de representar los trabajos y generar las planificaciones que se puede utilizar en un genético que resuelve el JSP. Esos mismos autores desarrollan en [8] un nuevo algoritmo genético utilizando un operador de inversión. Xu y Li en [159] proponen un método denominado IGA (Immune Genetic Algorithm), que es una combinación de la teoría de la inmunidad con un algoritmo genético, y demuestran su eficacia en la minimización del makespan en el JSP clásico. En [163], Zhang et al. proponen la utilización de un nuevo tipo de planificaciones, que es un subconjunto de las planificaciones activas, y que denominan *full active*. Las utilizan en un algoritmo genético hibridizado con búsqueda local, obteniendo buenos resultados. Wang et al. proponen en [154] un novedoso método de codificar los cromosomas; en dicho método las operaciones de cruce y mutación se efectúan en un espacio codificado en tres dimensiones. En [165], Zobolas et al. tratan la minimización del makespan en el problema job shop tradicional, y aplican una combinación de tres técnicas: un algoritmo basado en evolución diferencial para generar una población inicial de soluciones, y posteriormente una combinación de algoritmo genético y búsqueda local de tipo VNS (Variable Neighborhood Search) para tratar dicha población inicial.

En cuanto a trabajos sobre variantes del JSP, podemos citar a Candido et al., que en [34] consideran una extensión del JSP clásico con tiempos de setup y otras características de plantas de producción reales, como planes alternativos de procesamiento o recursos renovables. Los autores proponen un algoritmo genético hibridizado con reglas para la inicialización y un método simple de escalada para la mejora de los cromosomas. Utilizan una vecindad basada en cambiar el orden de operaciones pertenecientes al camino crítico que había sido previamente estudiada y formalizada para el JSP clásico. Obtienen vecinos de un modelo simplificado y después evaluan dichos vecinos en el problema real con setups. El hecho de considerar el camino crítico del problema simplificado implica que algunos de los vecinos no son soluciones que mejoran, porque se obtienen invirtiendo arcos que no están en el camino crítico del problema real. Además, se descartan muchos vecinos que podrían mejorar la solución actual del problema real. En [40] Cheung y Zhou utilizan un algoritmo genético para el SDST-JSP combinado con dos reglas, la primera esta basada en la suma

de tiempo de procesamiento y tiempo de setup más pequeño, y la segunda esta basada en la mayor cantidad de trabajo restante. Estas reglas se introducen en la función de evaluación del genético. Comparan el algoritmo genético con el heurístico propuesto por Choi y Korkmaz en [42] y obtienen mejores resultados. También, en [41], Choi y Choi consideran una variante del SDST-JSP con máquinas alternativas para las operaciones. Resuelven el problema a través de un algoritmo genético hibridizado con búsqueda local, y prueban el método sobre un conjunto de problemas definidos a partir de instancias del JSP clásico. En [61], González et al. han extendido dos métodos utilizados previamente para el JSP clásico, un algoritmo genético y un método simple de búsqueda local basado en cambios en el camino crítico. Los resultados experimentales sobre las instancias propuestas por Cheung y Zhou en [40] muestran que el algoritmo genético hibridizado con búsqueda local es mucho más eficiente que el algoritmo genético propuesto en [40]. El método propuesto por Essafi et al. en [49] consiste en un híbrido de algoritmo genético y búsqueda local para minimizar el weighted tardiness en el problema job shop clásico. La búsqueda local que utilizan se basa en inversiones de arcos en el camino crítico e itera entre fases de mejora y de perturbación. Las fases de mejora consisten en un método de escalada, mientras que las de perturbación eligen aleatoriamente un cierto número de movimientos de la vecindad, para poder escapar del máximo local encontrado. Realizan un estudio experimental bastante extenso y obtienen resultados muy competitivos. En [99] Naderi et al. proponen un algoritmo híbrido para la minimización del makespan en el SDST-JSP. Consiste en un algoritmo genético que incorpora algunas características adicionales, como una fase de reinicio y búsqueda local. En ese trabajo se evalúan también varios operadores genéticos y parámetros.

5.4. Representación de los individuos

Una de las claves del éxito de una búsqueda genética está en elegir una buena representación para los individuos. Esta representación debe permitir que los operadores genéticos produzcan individuos válidos y que éstos sean eficientemente evaluables. Dicha representación también será la base de la que dependen el resto de operadores, que deben ser diseñados para que utilicen de manera eficiente las propiedades de la representación elegida. La representación más simple de todas es la binaria, en la que cada cromosoma es simplemente un vector de ceros y unos. Sin embargo, se debe definir la representación que más convenga según el problema que se quiera resolver.

En el caso del JSP, un individuo debería representar una determinada planificación de todas las tareas del problema. Durante los últimos años se han propuesto las siguientes representaciones para este problema (véase el trabajo de Cheng et al. en [39]):

1. Operation-based representation
2. Job-based representation
3. Preference list-based representation
4. Job pair relation-based representation
5. Priority rule-based representation
6. Disjunctive graph-based representation
7. Completion time-based representation
8. Machine-based representation
9. Random keys representation

Estas representaciones se pueden clasificar en dos categorías básicas: directas e indirectas. En las *directas* una planificación se codifica en un cromosoma y el algoritmo genético se utiliza para evolucionar dichos cromosomas y encontrar una planificación mejor. Entre este tipo de representaciones se encuentran la 1, 2, 4, 7 y 9 de la lista anterior. En las *indirectas*, como por ejemplo la basada en reglas de prioridad, se codifica en cada cromosoma una secuencia de reglas de prioridad para asignar trabajos, pero no una planificación. Entonces, el algoritmo genético se utiliza para evolucionar los cromosomas y encontrar una mejor secuencia de reglas de prioridad. Entre este tipo de representaciones se encuentran la 3, 5, 6 y 8 de la lista anterior.

Además de todos estos tipos de codificación, recientemente Wang et al. proponen en [154] una representación novedosa que permite que los operadores de cruce y mutación se ejecuten en un espacio tridimensional, sin embargo en el estudio experimental no se comparan con ninguno de los métodos más representativos del estado del arte, y por tanto está por ver que sea realmente eficiente.

En general, el método más utilizado en la literatura es el basado en operaciones. Esta representación codifica una planificación como una secuencia de operaciones y cada gen del cromosoma representa una operación. La forma más natural es llamar a cada operación

mediante un número, tal y como se hace en la representación mediante permutaciones para el problema del viajante de comercio. Sin embargo, debido a la existencia de restricciones de precedencia, no todas las posibles permutaciones de números definen planificaciones factibles. Para resolver este problema, a continuación describiremos la alternativa más utilizada, ya introducida en la sección 2.6.

El esquema de permutaciones con repetición fue propuesto por Bierwirth en [27]. En este esquema, un cromosoma es una permutación del conjunto de operaciones, cada una de ellas representada por el número de su trabajo, con lo que el cromosoma es representado, de hecho, por una permutación con repetición de los trabajos. De esta forma cada número de trabajo aparece en un cromosoma tantas veces como número de operaciones tenga dicho trabajo. Por ejemplo, el cromosoma (2 1 1 3 2 3 1 2 3) representa realmente la permutación de operaciones ($\theta_{21}\,\theta_{11}\,\theta_{12}\,\theta_{31}\,\theta_{22}\,\theta_{32}\;\;\theta_{13}\,\theta_{23}\,\theta_{33}$). Esta permutación debe entenderse que expresa planificaciones parciales para cada conjunto de operaciones que requieren la misma máquina. Esta codificación presenta varias características interesantes; por una parte es sencillo ver que cualquier permutación representa una planificación factible, y por otra parte es fácil de evaluar con diferentes algoritmos y permite utilizar operadores genéticos eficientes. En [149], Varela et al. comparan esta codificación con otras codificaciones basadas en permutaciones y demuestran que esta es la mejor para el JSP en un estudio experimental sobre el conjunto de 12 instancias más frecuentemente utilizado. Por todos estos motivos, el esquema de permutaciones con repetición es el elegido en esta tesis para codificar los cromosomas.

5.5. Generación de la población inicial

El método más sencillo de crear la población inicial de individuos es generarlos aleatoriamente, siempre cuidando la factibilidad de los individuos generados. Utilizando la representación de permutaciones con repetición, un individuo factible para el SDST-JSP consiste simplemente en un cromosoma que tiene tantos números diferentes como número de trabajos tenga el problema, y tantas ocurrencias de cada uno como número de recursos tenga el problema.

Un método más avanzado, aunque más costoso en tiempo de ejecución, consiste en generar heurísticamente los individuos iniciales, para comenzar con una población inicial de individuos de mejor calidad. Sin embargo, algunos trabajos demuestran que este método no

siempre es buena idea. Por ejemplo, en [117] Puente et al. indican que una generación inicial heurística de la población funciona de forma eficiente sólo si las instancias son perfectas para dichos heurísticos, es decir, que funciona bien para algunas instancias pero no tanto en otras. También comentan que es aconsejable utilizar operadores genéticos adecuados, para tratar de preservar las características que los heurísticos implantaron en los cromosomas iniciales. Por otra parte, en [61] González et al. demuestran que generar la población inicial de forma heurística sólo interesa en instancias de gran tamaño.

También puede ser interesante que los individuos generados sean bastante diferentes entre sí, ya que una buena variedad genética en la población inicial suele ser muy beneficiosa para el desarrollo del algoritmo y contribuye a evitar una convergencia prematura en la ejecución. Sin embargo, esta última técnica suele ser bastante costosa computacionalmente.

Finalmente, en esta tesis hemos optado por el método menos costoso en tiempo de ejecución, es decir, generar la población inicial de forma aleatoria y sin tener en cuenta su diversidad.

5.6. Tamaño de la población

El tamaño de la población es una decisión importante en un genético, ya que si ésta es muy pequeña, el genético podría converger rápidamente, y si es muy grande quizás se desperdiciarían muchos recursos computacionales, al gastar mucho tiempo en la mejora de los individuos. En general, se pueden utilizar estrategias de población de tamaño fijo o variable. En esta tesis utilizaremos un tamaño fijo, que se elegirá dependiendo del tiempo de ejecución que deseemos emplear en el experimento, ya que este parámetro es uno de los factores más importantes que determinan el tiempo total de ejecución en un genético.

5.7. Operadores de selección

La fase de selección simula la supervivencia de los individuos más fuertes o mejor adaptados de una generación a otra. Esta selección no se debe hacer de forma completamente aleatoria, sino que los individuos con un fitness mayor deberían tener más posibilidades de ser elegidos para la siguiente generación. A continuación se detallan las principales técnicas de selección comúnmente utilizadas en los genéticos.

Una estrategia de selección por ruleta se basa en asignar a cada individuo una porción

de una "ruleta" virtual basándose en el fitness de cada uno. Así, los individuos peores tendrán una porción menor de la ruleta que los individuos mejores. A continuación se generan números aleatorios que simulan posiciones dentro de la ruleta, y se eligen los individuos correspondientes a dichas posiciones. En este tipo de estrategias de selección es frecuente utilizar técnicas de penalización y escalado, o de ranking, para evitar los problemas que puede causar que las diferencias entre el mejor individuo y el peor individuo sean demasiado grandes (lo que podría llevar a una convergencia prematura en la población) o demasiado pequeñas (lo que podría llevar a demasiada aleatoriedad en la búsqueda).

La estrategia de selección por torneo esta basada en el elitismo. La idea básica de este método es seleccionar en base a comparaciones directas entre individuos. La forma más común de torneo es elegir p individuos al azar de la población y seleccionar el mejor de todos ellos. El tamaño del torneo es p, y la elección de ese parámetro es un factor importante, ya que si es grande favorecerá demasiado a los mejores individuos de toda la población y nunca se elegirá a los no tan buenos, mientras que si es pequeño la probabilidad de elegir a un individuo malo será bastante alta. En el tipo de problemas de scheduling tratados en esta tesis, en los que los valores de fitness suelen ser muy grandes y no es fácil encontrar buenos parámetros de escalado, suele funcionar mejor la selección por torneo que la selección por ruleta. Hay un cierto número de variantes de la estrategia de torneo, y a continuación describimos la utilizada en esta tesis: disponiendo de las dos generaciones (padres e hijos) se seleccionan de entre cada grupo de dos padres y dos hijos a los dos mejores individuos (observar que en el operador de la ruleta y en otros operadores tradicionales, se puede elegir únicamente entre los hijos). Es decir, todos los cromosomas se agrupan aleatoriamente por parejas, y cada una de esas parejas se cruza para obtener dos descendientes. Después se elige que pasen a la siguiente generación los dos mejores individuos, de entre cada pareja de progenitores y su correspondiente pareja de hijos.

5.8. Operadores de cruce

Este operador permite el intercambio de información entre los cromosomas de la población actual, por lo que tiene un gran impacto en el resultado del algoritmo, y por ello es uno de los operadores genéticos que han recibido más atención por parte de los investigadores. Los operadores de cruce típicamente combinan las características de dos padres para formar dos hijos intercambiando fragmentos de sus progenitores, aunque existen operadores de cru-

ce que pueden cruzar entre sí a más de dos individuos. El cruce se realiza, por lo general, después de realizar la selección de individuos.

En el caso simple en el que la representación de los individuos es binaria, el operador de cruce más básico es el que se denomina "en un punto", y consiste en elegir una posición aleatoria en uno de los padres. El primer hijo tendrá los genes desde el 0 hasta la posición elegida del primer padre y el resto de genes del segundo padre. El segundo hijo, de forma equivalente, tendrá los genes desde el 0 hasta la posición elegida del segundo padre y el resto de genes del primer padre. Hay diversas variantes de este cruce básico, como son el cruce en dos puntos o el cruce uniforme, que consiste en elegir aleatoriamente para cada gen del hijo el padre del que se copia ese gen.

Sin embargo, en el caso de la representación mediante permutaciones con repetición no se puede utilizar un operador de cruce tan básico, ya que eso podría producir individuos no factibles. En esta tesis, para cruzar los cromosomas utilizaremos el *Job Order Crossover* (JOX) descrito por Bierwirth en [27], que fue diseñado específicamente para el JSP. Dados dos padres, JOX elige aleatoriamente un subconjunto de trabajos, y copia los genes de dichos trabajos al primer hijo en las mismas posiciones en las que estuvieran en el primer padre, y los restantes genes se toman del segundo padre en el mismo orden relativo en el que se encuentren en éste. Un segundo hijo se crea de la misma forma, pero intercambiando los roles de los padres. El siguiente ejemplo aclara el funcionamiento del operador JOX. Si se consideran los siguientes dos padres

Padre1 (**2** 1 1 3 **2** 3 1 **2** 3) Padre2 (3 3 1 **2** 1 3 **2** **2** 1)

El subconjunto de trabajos elegido aleatoriamente es el que esta marcado en negrita en los padres (trabajo 2), entonces la descendencia es

Hijo1 (**2** 3 3 1 **2** 1 3 **2** 1) Hijo2 (1 1 3 **2** 3 1 **2** **2** 3)

Por lo tanto, el operador JOX mantiene una subsecuencia de operaciones en las mismas posiciones en las que están en uno de los padres, mientras que las restantes operaciones mantienen el mismo orden relativo en las que están en el otro padre, pero sus posiciones en general cambian.

5.9. Operadores de mutación

El último paso, la mutación, tiene como objetivo cambiar de forma arbitraria alguno de los genes de un individuo con el fin de introducir diversidad en la población. Algunos investigadores sugieren utilizar una alta probabilidad de mutación al inicio de la búsqueda e ir decrementando exponencialmente dicha probabilidad, aunque lo más frecuente es utilizar una probabilidad constante durante toda la búsqueda pero inferior al 5 %. Otro ejemplo de aplicación del operador de mutación, propuesto por Zhang et al. en [163], es el siguiente: cuando eligen dos cromosomas para aplicar el operador de cruce, los cruzan con probabilidad 1 si su fitness es distinto, y sólo en el caso de que su fitness sea idéntico entonces les aplican el operador de mutación con una cierta probabilidad.

En esta tesis hemos optado por no utilizar ningún operador de mutación. Uno de los motivos es que utilizamos el operador de cruce JOX con probabilidad 1, y dicho operador puede cambiar cualesquiera dos operaciones que requieren la misma máquina; y esto proporciona un efecto de mutación implícita. Por otra parte, muchos investigadores piensan que el operador de mutación no juega un papel importante en el proceso evolutivo, en particular cuando el algoritmo genético se combina con búsqueda local. Por ejemplo, en [49] Essafi et al. miden el rendimiento de su algoritmo genético híbrido utilizando diversas probabilidades de un operador de mutación que intercambia dos elementos al azar del cromosoma. Experimentan con cuatro diferentes probabilidades de mutación (0.0, 0.05, 0.1 y 0.5), y llegan a la conclusión de que los resultados obtenidos sin utilizar mutación son tan buenos como los obtenidos con los otros valores.

Por estas razones no hemos utilizado ningún operador de mutación, y por lo tanto la selección de los parámetros para el estudio experimental se simplifica considerablemente, ya que la probabilidad de cruce se fija a 1 y la probabilidad de mutación no necesita ser especificada. Algunos experimentos sobre este tema se detallan en la sección 8.3.10 de esta tesis, en donde comprobaremos que no utilizar ningún operador de mutación es probablemente la mejor opción.

5.10. Operadores de fitness

El operador de decodificación o fitness determina la calidad de un individuo, y por lo tanto la probabilidad de supervivencia de un individuo en la fase de reproducción. Dicha evaluación no debe tener un coste computacional excesivo, porque será normalmente el ope-

rador que consuma la mayor parte del tiempo de ejecución del algoritmo genético, pero por otra parte debe discriminar bien la calidad de los individuos. Se puede utilizar simplemente un constructor de planificaciones semiactivas o activas o también, como ya se ha comentado, un método de búsqueda local simple o incluso búsqueda tabú.

Por otra parte, hay que tener en cuenta que cuando se aplica un constructor de planificaciones activas o una búsqueda local a un determinado cromosoma, será habitual que se intercambie el orden de algunas de las operaciones del cromosoma original. Hemos optado por copiar el nuevo orden generado a la secuencia del cromosoma, para que sus características se transmitan mejor a su descendencia. Este efecto de la función de evaluación se conoce como evolución Lamarckiana, como detallaremos en la sección 5.13.

5.11. Tipos de reemplazo y elitismo

La estructura del genético explicado en la sección 5.2 es generacional, pero también existen genéticos de sustitución inmediata o también llamados de estado uniforme. Son aquellos en donde los individuos generados por una pareja de padres entran directamente en la población y reemplazan a los miembros con menor aptitud, sin esperar a que se genere toda una nueva población.

Por otra parte, en los genéticos con sustitución generacional es común utilizar el operador de elitismo, que consiste en que el mejor o mejores individuos de la población pasan a la siguiente generación sin sufrir ningún cambio, cruce o mutación. Esto será útil para no perder una buena solución del problema, ya que por ejemplo si se utiliza una estrategia de selección por ruleta podría suceder que el mejor individuo de la siguiente generación sea peor que el mejor individuo que ya se tenía, lo cual no parece interesante. Sin embargo con la estrategia de selección utilizada en esta tesis (ver sección 5.7) es sencillo ver que el mejor individuo de una generación siempre será igual o mejor que el mejor individuo de la generación anterior, por lo que no será necesario un operador de elitismo.

5.12. Ordenación topológica

Al hablar de la representación de una solución mediante vectores en la sección 2.6 hemos comentado que varios vectores pueden dar lugar a la misma planificación final. Por el contrario, planificaciones diferentes siempre darán lugar a vectores diferentes. Por estos

motivos, se podría pensar que la forma de convertir una matriz en un vector puede influir en la evolución del algoritmo genético.

En principio sí que puede haber diferencias en la ejecución, ya que puede ocurrir que el operador de cruce produzca planificaciones diferentes al cruzar dos vectores padre formados de diferente forma, aunque dichos vectores representen la misma planificación. A continuación se muestra un ejemplo de este último concepto, utilizando el operador de cruce JOX descrito en la sección 5.8.

Supongamos que en una determinada instancia compuesta por 3 trabajos y 3 tareas, las tareas θ_{11}, θ_{21} y θ_{32} deben ejecutarse en la máquina M_1, las tareas θ_{12}, θ_{23} y θ_{31} deben ejecutarse en la máquina M_2, y las tareas θ_{13}, θ_{22} y θ_{33} deben ejecutarse en la máquina M_3. Entonces, es sencillo comprobar que las dos siguientes ordenaciones topológicas, expresadas mediante permutaciones con repetición, corresponden a la misma planificación:

Padre 1A: 2 1 3 1 2 3 1 3 2

Padre 1B: 3 2 1 1 2 2 1 3 3

Sin embargo, si cruzamos esos individuos con este otro cromosoma:

Padre 2: 3 2 3 3 2 2 1 1 1

Utilizando el operador de cruce JOX, y eligiendo aleatoriamente el trabajo 2, entonces la descendencia del padre 1A con el padre 2 es:

2 3 3 3 **2** 1 1 1 **2**

1 **2** 3 1 **2** **2** 3 1 3

Y la descendencia del padre 1B con el padre 2 es:

3 **2** 3 3 **2** **2** 1 1 1

3 **2** 1 1 **2** **2** 1 3 3

Es fácil comprobar que los cuatro hijos representan cuatro planificaciones diferentes entre sí, aunque la planificación que representa el primer hijo del padre 1A con el padre 2 es idéntica a la del padre 2, y la del segundo hijo del padre 1B con el padre 2 es idéntica a la del padre 1B. Todo esto ocurre a pesar de que padre 1A y padre 1B representaban la misma planificación.

Teniendo en cuenta este comportamiento, es posible que para un algoritmo genético sea una buena opción no realizar los ordenamientos topológicos siempre de la misma forma,

ya que eso podría limitar la variabilidad en la población. Entonces, podemos enfocar la construcción del vector a partir de la planificación de estas formas:

- *De forma fija:* en cada paso, se comprueba qué tareas se pueden planificar en ese momento y se elige una de ellas según un determinado criterio (por ejemplo la que tenga el número de trabajo o de recurso más bajo, o la que lleve más o menos tiempo a la espera de entrar en el vector).

- *De forma aleatoria:* en cada paso, se comprueba qué tareas se pueden planificar en ese momento y se elige una de ellas aleatoriamente.

En la sección 8.3.12 mostramos algunos experimentos sobre este tema y veremos que los resultados obtenidos por los dos métodos son muy similares, y por lo tanto por sencillez resulta mejor opción en general realizar la ordenación topológica de forma fija.

5.13. Combinación con una búsqueda local

Los algoritmos genéticos, como ya hemos comentado, son una de las metaheurísticas bio-inspiradas más populares entre los investigadores en el área de la optimización combinatoria. Una de las principales razones de su éxito es que no requieren demasiado conocimiento sobre el dominio del problema para obtener soluciones razonablemente buenas; de hecho el único conocimiento que se explota suele estar dentro de la función de evaluación. Al mismo tiempo, este hecho representa la principal debilidad de los algoritmos genéticos. Es bien sabido que la calidad de las soluciones que alcanza un algoritmo genético simple en problemas complejos no suele ser muy buena, precisamente por esta falta de conocimiento. Por esta razón, frecuentemente se combinan con otras técnicas que les permite llegar a regiones del espacio de búsqueda muy buenas en problemas complejos; regiones que son muy difíciles de alcanzar por un algoritmo genético simple que utiliza únicamente operadores independientes del problema.

La eficiencia de la búsqueda se puede mejorar si la adaptación evolutiva en un algoritmo genético se combina con algún mecanismo de aprendizaje durante la vida de cada individuo. Esta combinación se inspira en el concepto de meme, creado por Dawkins en [44]. Un meme es, en las teorías sobre la difusión cultural, la unidad teórica de información cultural transmisible de un individuo a otro o de una mente a otra (o de una generación a la siguiente). En optimización evolutiva, un meme a menudo consiste en una búsqueda local, y el

algoritmo resultante se suele denominar algoritmo memético, como indica Moscato en [96], aunque en la literatura también se han utilizado otros términos, como algoritmos genéticos híbridos, búsquedas locales genéticas, o algoritmos culturales. Los algoritmos meméticos se han estudiado bastante y se han utilizado con éxito en una gran variedad de problemas

En [78] Krasnogor y Smith estudian varias aplicaciones de los algoritmos meméticos, proponen un modelo sintáctico para dichos algoritmos, y también discuten las decisiones más relevantes que se deben tener en cuenta para diseñar un algoritmo memético lo más eficiente posible. Por ejemplo cuándo y dónde se debe aplicar la búsqueda local dentro del ciclo evolutivo, o cuánto peso computacional se le debe dedicar a la búsqueda local. Por supuesto, hay muchas posibles respuestas a estas preguntas, y la mejor opción se puede deducir a través de razonamientos teóricos, pero más frecuentemente se elige mediante un estudio experimental.

En esta tesis hemos decidido aplicar la búsqueda local a la planificación de todos y cada uno de los cromosomas de la población. Por otra parte, en el caso de utilizar una búsqueda local simple de tipo escalada continuaremos la búsqueda local hasta alcanzar un óptimo local, y en el caso de utilizar una búsqueda local más avanzada que sea capaz de escapar de óptimos locales, como por ejemplo la búsqueda tabú, le asignaremos un número máximo de iteraciones que elegiremos mediante algunos experimentos previos. Cuando se combina un algoritmo genético y una búsqueda local, para mantener el tiempo de ejecución en valores razonables lo habitual es reducir el coste computacional del proceso evolutivo disminuyendo el número de individuos en la población y el número de generaciones.

Otra decisión importante es el modelo de evolución del algoritmo memético. En este punto hay dos posibilidades: evolución Lamarckiana o evolución Baldwiniana. El efecto Baldwin ocurre cuando una característica se hace innata en la población como resultado de ser aprendida en la sociedad. Una de las razones de que ocurra este efecto puede ser que la selección genética hace que los individuos sean mejores cada vez en aprender esa característica de la sociedad. Es decir, que realmente la característica no esta codificada en los cromosomas de los individuos, sino que lo que esta codificado es la capacidad de aprenderla. Exista o no exista el efecto Baldwin en la evolución natural, se puede implementar fácilmente en la evolución artificial de la siguiente forma: cada individuo sufre un proceso de mejora mediante una búsqueda local, y entonces el fitness del individuo es aquel del individuo mejorado, en lugar de ser el fitness antes de la mejora.

Por otra parte, la evolución Lamarckiana ocurre cuando los cambios de un individuo

a lo largo de su vida se traspasan a su descendencia. Esto ocurrirá si las características adquiridas por el individuo se codifican de nuevo en la estructura del cromosoma. Es bien sabido que este efecto no ocurre en la evolución natural, pero frecuentemente se implementa en la evolución artificial, ya que permite que las características adquiridas por los padres (por ejemplo a través de la búsqueda local) se traspasen de forma inmediata a su descendencia. Uno de los inconvenientes de este modelo es que puede afectar negativamente a la diversidad del material genético en la población.

Los efectos Baldwiniano y Lamarckiano se pueden utilizar en un algoritmo memético, e incluso se pueden combinar en diferentes proporciones. En principio utilizaremos el modelo de evolución Lamarckiano, ya que en la literatura se puede ver que es el más ampliamente utilizado y que suele funcionar mejor que el modelo Baldwiniano. Sin embargo en la sección 8.3.8 comprobaremos esto de forma experimental realizando varias pruebas comparando los dos modelos. Para simplificar ese estudio experimental optaremos por considerar los dos casos extremos, es decir evolución Baldwiniana pura y evolución Lamarckiana pura.

5.14. Conclusiones

En esta sección hemos profundizado en la estructura de los algoritmos genéticos que utilizaremos a lo largo de esta tesis. Después de una breve introducción y una revisión bibliográfica sobre su aplicación a problemas de scheduling, nos hemos adentrado en su estructura general y posteriormente en cada uno de sus componentes. Desde la representación de las soluciones y la generación de la población inicial y su tamaño, hasta los operadores genéticos de selección, fitness, cruce y mutación. Para que el algoritmo genético sea eficaz debe haber una cierta sinergia entre todos sus elementos y operadores. Este tipo de algoritmos evolutivos ha resultado ser muy eficaz a lo largo de los años en la resolución de una gran variedad de problemas, incluídos los de scheduling. Para concluir el capítulo explicamos cómo se combina un algoritmo genético con una búsqueda local para construir un algoritmo memético, y comentamos los dos modelos evolutivos más utilizados en dichos algoritmos.

Capítulo 6

LA BÚSQUEDA LOCAL

6.1. Introducción

La búsqueda local es un método con bastante éxito en la inteligencia artificial. Los algoritmos de búsqueda local realizan una búsqueda en un espacio de soluciones potenciales del problema tratando de encontrar aquella que maximice o minimice una determinada función objetivo. Esta función objetivo suele tener una forma muy irregular, con numerosos máximos y mínimos locales, lo cual hace que la búsqueda del óptimo global sea muy costosa. Normalmente la búsqueda local no mantiene traza de todo el proceso realizado hasta el momento, sino que sólo utiliza información sobre el estado actual y sus vecinos. A pesar de esto, la búsqueda local es una estrategia muy eficaz en muchos casos.

Además se puede combinar con otras estrategias, como por ejemplo con los algoritmos genéticos descritos en la sección anterior. Los algoritmos genéticos son muy efectivos realizando una búsqueda global pero a menudo sufren de convergencia prematura, mientras que los métodos de búsqueda local son muy buenos refinando la búsqueda pero a menudo se estancan en óptimos locales. Es decir, que los dos métodos se complementan a la perfección, y por este motivo en la literatura se ha tratado en numerosas ocasiones de hibridizar estas dos metaheurísticas.

6.2. Estructura general de una búsqueda local

A grandes rasgos, la búsqueda local se implementa definiendo la vecindad de una solución del espacio de búsqueda como el conjunto de soluciones a las que se puede llegar mediante una regla de transformación dada. Una vez que se construye la vecindad se debe evaluar alguno o todos los vecinos. Esta es una de las acciones más críticas del algoritmo por el elevado tiempo de ejecución que puede suponer, especialmente si el número de soluciones vecinas es muy grande, o bien si el procedimiento de evaluación de cada solución es muy costoso. A continuación se selecciona una de las soluciones vecinas con un determinado criterio, normalmente la que tiene menor coste. Por último, la solución actual se reemplaza por la solución vecina elegida, siempre y cuando cumpla el criterio de aceptación, que puede consistir simplemente en que la solución vecina elegida sea mejor que la solución anterior. La búsqueda local finaliza al cabo de cierto número de iteraciones, o bien cuando ningún vecino satisface el criterio de aceptación. El algoritmo 6.1 muestra la estrategia típica de un método de búsqueda local.

Algoritmo 6.1 El esquema básico de una búsqueda local

Requiere: Una planificación inicual H y una instancia de problema P

Produce: Una planificación que se intentará que sea mejor que H

1. Evaluar la planificación H;

mientras No se cumpla el criterio de parada **hacer**

2. Generar la vecindad de H con algún método N, $N(H)$;
3. Elegir $H' \in N(H)$ con el criterio de selección;
4. Reemplazar H por H' si el criterio de aceptación se cumple;

fin mientras

devuelve La mejor planificación encontrada hasta el momento;

En este esquema tan básico hay varios puntos que hay que precisar: la forma de generar la solución inicial, la estrategia que se debe utilizar para generar los vecinos de una determinada solución, los criterios de selección y aceptación, y la condición de parada del bucle. En las sucesivas secciones de este capítulo se explicará cada uno de estos puntos, excepto el tema de generar los vecinos, en el cual se profundizará en el capítulo 7. A continuación explicaremos la generación de la solución inicial, las posibles condiciones de parada, y después detallaremos los diferentes criterios de selección y aceptación que se pueden encontrar en la literatura, a

la vez que hacemos una revisión bibliográfica según los criterios utilizados.

6.2.1. Generación de una solución inicial

Este punto se puede abordar de la forma más simple o más complicada que se desee. La forma más simple es generar una solución aleatoria, y las más complicadas pasan por utilizar heurísticos que generen una solución inicial que ya sea lo mejor posible.

La utilización de heurísticos depende completamente del problema que se desee resolver. La gran mayoría de los problemas dispone de algún heurístico efectivo para la generación de una solución inicial aceptable. También es importante que el coste computacional de obtener la solución inicial no sea demasiado elevado, ya que en ese caso el tiempo utilizado en obtenerla podría no compensar la ventaja ganada durante la ejecución del algoritmo.

Por el contrario, utilizar una solución inicial aleatoria tiene la ventaja de su simplicidad y rapidez. Sin embargo, lo más probable es que la calidad de dicha solución sea bastante mala, y eso puede repercutir negativamente en la ejecución del algoritmo. Por una parte, se necesitará un mayor número de iteraciones para llegar a un resultado aceptable, y por otra parte es posible que la solución generada corresponda a una porción del espacio de búsqueda en la que no haya soluciones buenas, y además sea complicado salir de dicha porción del espacio.

Sin embargo, cuando el algoritmo de búsqueda local es más complejo suele tener menos relevancia la calidad de la solución inicial. Por ejemplo, Zhang et al. en [162] indican que la calidad de la solución inicial no tiene apenas influencia en los resultados del algoritmo de búsqueda tabú que proponen. En esta tesis también optamos por generar la solución inicial de forma aleatoria cuando utilicemos un método de búsqueda local por separado. Cuando se utiliza la búsqueda local en combinación con un algoritmo genético, lo habitual es que ésta se aplique a cada uno de los cromosomas del algoritmo genético, y por tanto los cromosomas serán las soluciones iniciales en ese caso.

6.2.2. Condición de parada

La condición de parada depende en gran medida de los criterios de selección y aceptación de la búsqueda local, los cuales se explicarán con detalle en la siguiente sección.

Si se utiliza una estrategia de escalada simple o de máximo gradiente, la condición de parada más lógica parece ser encontrar un óptimo local, ya que por una parte esas estrategias no permiten escapar de óptimos locales, y por otra parte detener la búsqueda antes de llegar

a un óptimo local no parece aconsejable porque si continuamos la búsqueda podríamos seguir mejorando la solución actual.

Si se utilizan estrategias más avanzadas, como enfriamiento simulado o búsqueda tabú, no se puede utilizar una condición de parada tan simple. Algunas de las posibilidades son las siguientes: especificar un número máximo de iteraciones para el algoritmo, especificar un tiempo máximo de procesamiento, especificar un valor de la función objetivo que se desea alcanzar (por ejemplo si conocemos el valor del óptimo o una cota inferior), o especificar un número máximo de iteraciones entre dos mejoras sucesivas (es decir que la ejecución se acabaría si transcurre un cierto número de iteraciones sin mejorar la solución desde la última mejora obtenida).

6.3. Criterios de selección y aceptación

Se pueden utilizar diferentes estrategias para elegir uno de los vecinos. Las más simples son las de tipo "escalada", que consisten en ir eligiendo vecinos que mejoran al individuo actual, hasta llegar a un óptimo local, es decir un punto del espacio de búsqueda en el cual todos los vecinos son iguales o peores que la solución actual. Como ejemplos se pueden citar la escalada simple y la escalada de máximo gradiente. La escalada simple evalúa vecino a vecino, y en cuanto encuentre uno mejor que la solución actual, se sustituye la solución actual por dicho vecino. La escalada de máximo gradiente evalúa todos los vecinos y sustituye la solución actual por el mejor de todos los vecinos, si es que dicho vecino mejora al individuo actual.

La principal ventaja de los métodos de escalada es su rapidez y simplicidad. Su principal inconveniente es que se quedan atascados en óptimos locales, ya que si todos los vecinos son peores que la solución actual se termina la búsqueda, y dichos óptimos locales no tienen por qué ser necesariamente buenas soluciones. Para resolver este problema se han planteado muchas alternativas, pero los tres métodos más conocidos y utilizados en la literatura son las siguientes: permitir movimientos que pueden empeorar la solución actual, modificar la estructura de entornos, o volver a comenzar la búsqueda desde otra solución. A continuación se detallará cada uno de estos tres métodos.

6.3.1. Permitir movimientos que pueden empeorar la solución actual

Si se permiten movimientos que puedan empeorar la solución actual es claro que se puede escapar de un óptimo local. Hay varias formas de afrontar este tipo de estrategia. Las dos más conocidas son probablemente el enfriamiento simulado y la búsqueda tabú, aunque también detallaremos la denominada large step random walk y otros métodos.

El concepto de **enfriamiento simulado** (o *simulated annealing*) fue introducido por Kirkpatrick et al. en [76] y esta basado en el trabajo de Metropolis et al. en el campo de la termodinámica estadística presentado en [92]. Simula el proceso de enfriado en un sistema de partículas a medida que decrece la temperatura. La idea consiste en admitir con una cierta probabilidad algunos vecinos que empeoren a la solución actual, y de este modo poder escapar de óptimos locales. La principal diferencia de este algoritmo con respecto al método de escalada es la siguiente: el enfriamiento simulado realiza una selección aleatoria entre los vecinos de la solución actual. Si la solución elegida es mejor que la actual, siempre se acepta, al igual que en el método de escalada. Pero si la solución vecina es peor que la actual, la nueva solución se acepta con una determinada probabilidad que depende de dos parámetros: la temperatura T y el incremento de energía ΔE. Al principio de la búsqueda, la temperatura tiene un valor alto, inicialmente T_0, de modo que la probabilidad de aceptar una solución peor que la actual es alta. A medida que la búsqueda progresa, el valor de T se va actualizando de forma $T = \alpha(t, T)$, siendo t la iteración actual y α una función que decrece al aumentar t. De este modo, la probabilidad de aceptar una solución que empeora a la actual va disminuyendo a medida que avanza la búsqueda, hasta que al final únicamente se admiten soluciones que mejoren o igualen a la actual. El principal inconveniente que presenta este método es que requiere un ajuste de parámetros adecuado y el éxito o fracaso del método depende en gran medida de dicho ajuste. Normalmente, este ajuste depende fuertemente del problema y hay que realizarlo de forma experimental. Una ventaja que tiene este método es que resultados teóricos basados en la teoría de las cadenas de Markov han demostrado que con un programa de enfriamiento infinitamente lento, el algoritmo converge al óptimo global con probabilidad 1, a medida que la temperatura tiende a 0. Los detalles pueden ser consultados en el trabajo de Van Laarhoven y Aarts en [147] o en el de Aarts y Korst en [1]. Sin embargo, no se puede garantizar esa convergencia en programas de enfriamiento finitos. De hecho, Van Laarhoven y Aarts han demostrado que para garantizar llegar a una solución que esté a una distancia arbitrariamente próxima a la óptima, son necesarios tiempos de

computación de tipo exponencial. De todas formas, el enfriamiento simulado es un método que ha obtenido buenos resultados para una gran variedad de problemas combinatorios. En cuanto a su aplicación a problemas de scheduling, las primeras veces que se ha aplicado al JSP puede ser el algoritmo de Matsuo et al. en [87], el de Van Laarhoven et al. en [148] o el de Aarts et al. en [2]. Ejemplos mucho más recientes de su aplicación en diferentes problemas de optimización pueden ser los trabajos de Wu et al. en [158], de Rodriguez-Tello et al. en [120] o de Naderi et al. en [101]. En cuanto a aplicaciones recientes al SDST-JSP, en [98] Naderi et al. tratan de minimizar el makespan con un método de enfriamiento simulado. Realizan un estudio experimental con el objetivo de elegir los parámetros más adecuados para su algoritmo, y posteriormente se comparan con otros métodos en instancias derivadas de las instancias clásicas de Taillard. Una de las novedades es que cuando el algoritmo lleva un cierto número de iteraciones sin mejorar la mejor solución alcanzada hasta el momento, cambian a una estructura de vecindad que realiza movimientos un poco más amplios, generan 50 vecinos con ella, y continúan la búsqueda a partir del mejor de los 50 vecinos generados.

El **large step random walk** fue introducido por primera vez por Martin et al. en [85]. La idea consiste en iterar sucesivamente entre fases de diversificación (llamadas large step o paso largo) e intensificación (llamadas small step o paso corto). En los pasos largos se intenta guiar la búsqueda hacia nuevas regiones prometedoras del espacio de búsqueda, y éstas son exploradas en detalle mediante los pasos cortos. La solución inicial en la fase de paso corto es siempre la última solución alcanzada en la fase de paso largo anterior, mientras que la solución inicial en la fase de paso largo es la solución final obtenida en la fase de paso corto anterior. Por este motivo, las soluciones alcanzadas en iteraciones anteriores no se pierden totalmente en las siguientes iteraciones, tal como suele ocurrir en los métodos multiarranque. En los pasos largos la solución actual se modifica en mayor medida que en los pasos cortos. Por ejemplo, en [79] Kreipl desarrolla un método de búsqueda local basado en este algoritmo, y trata de minimizar el weighted tardiness en el problema JSP clásico. Los pasos largos utilizan el algoritmo de Metropolis por lo que se pueden aceptar soluciones peores que la actual y escapar de óptimos locales, mientras que los pasos cortos utilizan un algoritmo de escalada por lo que siempre alcanzan un óptimo local. Utiliza una estructura de vecindad previamente desarollada por Suh en [138], basada en el concepto de camino crítico, y compara sus resultados con los obtenidos por el heurístico shifting bottleneck de Singer y Pinedo en [133], obteniendo mejores resultados en general. Más recientemente, en [155] Wang et al. proponen un método similar para minimizar el makespan en el problema job-shop clásico.

Generan la solución inicial mediante la regla de prioridad Shortest Processing Time (SPT), y utilizan una estructura de vecindad basada en inversiones de arcos críticos. En los pasos largos aplican una modificación más profunda de la solución actual, y en los pasos cortos aplican un método de enfriamiento simulado. Si al final del enfriamiento simulado realizado en el paso corto mejoran la mejor solución encontrada hasta el momento en la búsqueda, aceptan esa solución, y en caso contrario la rechazan y vuelven a la solución anterior.

La **búsqueda tabú** fue introducida por Glover en [56] y se basa en un mecanismo de memoria, y con ello evita la elección de algunos vecinos dependiendo de la historia reciente de la búsqueda, o de la frecuencia con la que se realizaron algunas transformaciones para llegar al estado actual. En el caso más simple, el mecanismo de memoria se realiza mediante una estructura que registra las transformaciones que dieron lugar a las últimas soluciones, de modo que estas transformaciones no se consideran en la generación de los vecinos de la solución actual (se consideran tabú). El tamaño de la lista tabú debe ajustarse en función de las características del problema. Por otra parte, es posible que en algunos casos un movimiento de la lista tabú produzca en la solución actual una mejora, por este motivo se introduce lo que se denomina criterio de aspiración y que consiste en establecer excepciones a lo que indica la lista tabú. Al igual que en el enfriamiento simulado, uno de los inconvenientes de la búsqueda tabú es el ajuste de parámetros, por ejemplo el tamaño de la lista tabú, la elección de los movimientos que se deben registrar en dicha lista, la definición del criterio de aspiración, etc. De todas formas, la búsqueda tabú se ha empleado con éxito en muchos problemas, tanto utilizada por separado como en combinación con otras estrategias como por ejemplo los algoritmos evolutivos. En [142] Vaessens et al. hacen un estudio sobre algoritmos de búsqueda local, y demuestran que los algoritmos de búsqueda tabú hasta ese momento ofrecen mejores resultados que los algoritmos genéticos y que los métodos de enfriamiento simulado. Ya que la búsqueda tabú es uno de los principales métodos de resolución utilizados en esta tesis, haremos un análisis más detallado de ella y una revisión bibliográfica en la sección 6.4.

Existen **otros métodos** diferentes para escapar de óptimos locales permitiendo movimientos que empeoran la solución actual, algunos de ellos más simples que los ya explicados. Por ejemplo, Mati et al. en [86] proponen un algoritmo de búsqueda local que básicamente es un método de escalada de máximo gradiente que cuando alcanza un óptimo local realiza un cierto número de movimientos aleatorios (y que pueden empeorar la solución actual) para escapar de dicho óptimo. El algoritmo que proponen es capaz de minimizar cualquier

función objetivo regular, es decir, cualquier función objetivo que verifique que sea creciente según el tiempo de fin de los trabajos, y por tanto que siempre sea óptimo comenzar un trabajo lo antes posible. La estructura de vecindad que proponen esta basada en invertir un único arco crítico en cada movimiento, y utilizan estimaciones para evaluar a los vecinos y elegir uno de ellos.

6.3.2. Modificar la estructura de entornos

La **VNS** (Variable Neighborhood Search) es una metaheurística que se puede utilizar para realizar una búsqueda local. En su forma más básica consiste en que cuando se alcanza un óptimo local para una estructura de vecindad, se cambia la estructura de vecindad para intentar salir de dicho óptimo. El algoritmo continúa hasta que se haya encontrado una solución que sea óptimo local simultáneamente para todas las estructuras de vecindad consideradas. A continuación enumeramos tres observaciones realizadas por Hansen y Mladenovic en [66], por las que este método puede dar buenos resultados:

1. Un óptimo local con respecto a una estructura de vecindad no es necesariamente un óptimo local para otra estructura de vecindad diferente.

2. Un óptimo global es un óptimo local con respecto a todas las posibles estructuras de vecindad.

3. Para muchos problemas, los óptimos locales, con respecto a una o varias estructuras de vecindad están relativamente cerca unos de otros.

Esta última observación es empírica, e implica que un óptimo local a menudo ofrece algo de información sobre el óptimo global. Por ejemplo diversas variables con el mismo valor en los dos óptimos, aunque no se puede saber qué variables son. Por este motivo es interesante estudiar más en profundidad un óptimo local alcanzado para ver si encontramos otro óptimo local cercano mejor.

La versión más básica de la VNS requiere ajustar pocos parámetros, y produce soluciones de buena calidad, a pesar de que es mucho más sencillo que otros tipos de estrategias de búsqueda local. Aunque para utilizar este esquema se deben tomar las siguientes decisiones: número de estructuras de vecindad que se van a utilizar, cuáles serán dichas estructuras, y en qué orden habrá que explorarlas. En principio, lo mejor es utilizar primero las vecindades más simples y que consuman menos tiempo, y utilizar después otras vecindades cada vez más

complejas, cuando las primeras hayan llegado a un óptimo local para ellas. El inconveniente obvio de este algoritmo es que se también se quedará atascado en un óptimo local (aunque tenga que ser un óptimo local para todas las estructuras de vecindad consideradas), pero esto se podría resolver combinando este método con otro, por ejemplo con alguna estrategia que permita elegir movimientos que empeoren a la solución actual.

Como ejemplo de aplicación reciente, en [121] Roshanaei et al. proponen un método VNS para minimizar el makespan en el SDST-JSP. Generan la solución inicial mediante la regla de prioridad Shortest Processing Time (SPT). Representan una planificación mediante un vector en el que las tareas están en forma de permutaciones con repetición. Proponen tres estrategias de vecindad basadas en inserciones de tareas en otro lugar del vector, cada una de las estructuras modifica el vector en mayor medida que la estructura anterior. Aplican en principio la estructura más simple, y si no obtienen mejora van pasando a las estructuras más complejas, volviendo a la estructura de vecindad simple en cuanto logran una mejora. Otro ejemplo reciente lo tenemos en [100], en donde Naderi et al. proponen un método VNS muy similar al propuesto por Roshanaei et al. en [121] pero en este caso se aplica al problema Flexible Flow Lines con sequence-dependent setup times y con tiempos de mantenimiento en las máquinas.

6.3.3. Volver a comenzar la búsqueda desde otra solución

Por último, otra opción posible es anotar el óptimo local al que se ha llegado con un método tradicional de escalada, y comenzar otra búsqueda local nueva desde otra solución inicial diferente, o bien desde una de las soluciones intermedias alcanzadas durante la ejecución. De esta forma probablemente se alcanzará otro óptimo local distinto, y así, después de un número de intentos probablemente se llegará a una solución aceptable, sobre todo si el número de óptimos locales del problema no es muy alto. Ejemplos de este tipo de estrategia son las búsquedas multiarranque o el GRASP.

La **búsqueda multiarranque** se ha aplicado a una gran variedad de problemas, entre otros a los de optimización combinatoria. Estos métodos normalmente no presentan propiedades de convergencia al óptimo global del problema, y su aplicación se limita al campo de los algoritmos aproximados o heurísticos. Una búsqueda multiarranque suele tener dos fases diferenciadas: la construcción de una solución y la aplicación de un método de búsqueda sobre dicha solución. La fase de construcción en ocasiones se limita a la simple generación aleatoria de soluciones, mientras que otras veces se emplean complejos métodos de cons-

trucción que consideran las características del problema para obtener soluciones iniciales de buena calidad. También es frecuente utilizar métodos que aseguren que las soluciones iniciales generadas no sean demasiado parecidas unas de otras, para asegurar una cierta variedad en los óptimos alcanzados en cada iteración. En cuanto al método de búsqueda local, se pueden considerar tanto algoritmos simples de escalada o combinaciones con otros mecanismos para poder escapar de óptimos locales. También es posible utilizar información sobre las soluciones de inicio y sobre los óptimos locales encontrados para dirigir la búsqueda. De todas formas, se aplican estas dos fases del algoritmo hasta que se cumpla una condición de parada, que puede consistir simplemente en un número máximo de iteraciones, aunque también se han desarrollado en algunos trabajos reglas de parada de tipo bayesiano bastante complejas.

En el trabajo de Philip en [113] tenemos un ejemplo de aplicación de búsqueda local multiarranque a problemas de scheduling. Se trata de la minimización del makespan en una generalización del problema SDST-JSP que permite trabajos reentrantes y la utilización de varios recursos paralelos. Se utiliza un método de búsqueda local multiarranque cuyas soluciones iniciales se obtienen mediante simulación, y se comparan los resultados con los obtenidos por algunas reglas de prioridad, resultando mejor la búsqueda local en la mayoría de los casos.

El **GRASP** (*Greedy Randomized Adaptive Search Procedure*) es uno de los métodos multiarranque más aplicados actualmente. Al igual que el resto de métodos multiarranque, GRASP consta de dos fases: la de construcción y la de búsqueda. Aunque hay algunas variantes, el funcionamiento de un algoritmo GRASP típico es que en cada iteración de la fase de construcción se mantienen un conjunto de elementos candidatos que pueden ser añadidos a la solución parcial que se está construyendo. Todos los elementos candidatos se evalúan utilizando una función que mide su atractivo. En lugar de seleccionar el mejor de todos los elementos, se construye la lista restringida de candidatos, RCL (restricted candidate list), con un cierto número de los mejores individuos según una determinada función (esta es la parte *Greedy* o voraz del método). El elemento que finalmente se añade a la solución parcial actual se escoge al azar del conjunto RCL (esta es la parte *Randomized* o aleatoria del método). Entonces se recalcula la lista de elementos candidatos y se realiza una nueva iteración (esta es la parte *Adaptive* o adaptativa del método). Estos pasos se repiten hasta que se obtiene una solución del problema. A ésta se le aplica el método de mejora, que puede consistir en una búsqueda local simple, o en híbridos con búsqueda tabú

o enfriamiento simulado, entre otros. El método repite las fases de construcción y de mejora hasta que se cumpla el criterio de parada. Una de las principales desventajas del GRASP puro es su falta de estructuras de memoria, es decir, las iteraciones son independientes unas de otras y no utilizan información obtenida durante iteraciones previas. Uno de los remedios más utilizados para resolver este inconveniente es la utilización del **reencadenamiento de trayectorias** (*path relinking*) como método de post-optimización. Este método consiste en elegir pares de óptimos locales que haya alcanzado el algoritmo GRASP, y tratar de construir un "camino de soluciones" para llegar de una a otra, realizando un cambio simple en cada elemento o iteración del camino. En cada iteración del camino suele haber varios posibles movimientos para llegar a la solución destino, y en ese caso se elige al mejor de todos ellos. La idea de este método es que es muy posible que el espacio de búsqueda entre dos soluciones muy buenas contenga también soluciones muy buenas o incluso mejores.

Como ejemplo de aplicación de este método a un problema de scheduling, en [11] Armentano y Filho proponen un algoritmo GRASP para minimizar el total tardiness en problemas con máquinas paralelas y tiempos de setup. En la fase de construcción utilizan una memoria que guarda las mejores soluciones encontradas hasta el momento, con el objetivo de guiar el algoritmo voraz, prefiriendo parámetros que aparezcan mucho en esa lista. El equilibro entre diversificación e intensificación en la búsqueda lo controlan mediante un parámetro que regulan entre 0 (construcción completamente aleatoria) y 1 (construcción completamente determinista). La vecindad utilizada en la búsqueda local se basa en dos movimientos: 1) Insertar un bloque de hasta 3 trabajos consecutivos en otra máquina o en otra posición de la misma máquina, 2) Intercambiar bloques de hasta 3 trabajos consecutivos entre dos máquinas diferentes. La idea es ir alternando las dos vecindades hasta que se llega a un mínimo local respecto a las dos. Por último, como método de post-optimización, realizan un path relinking sobre la lista de las mejores soluciones obtenidas.

6.4. Búsqueda Tabú

6.4.1. Introducción

La búsqueda tabú (*Tabu Search*), que denotaremos TS en adelante, es un método avanzado de búsqueda local que puede escapar de óptimos locales eligiendo vecinos que son iguales o peores que la solución actual. Para evitar llegar de nuevo a soluciones que se han explorado recientemente, y también para explorar nuevas regiones del espacio de búsqueda,

la búsqueda tabú utiliza una estructura llamada lista tabú, que contiene un conjunto de movimientos que no están permitidos al generar la nueva vecindad.

6.4.2. Revisión bibliográfica sobre búsqueda tabú en problemas de scheduling

La búsqueda tabú fue introducida por primera vez por Glover en [56]. En el libro [59] Glover y Laguna realizan una revisión exhaustiva sobre el método. A continuación se comentan algunos de los métodos de búsqueda tabú más conocidos aplicados a problemas de scheduling. La mayoría de ellos tratan de minimizar el makespan en el job shop clásico sin tiempos de setup. Sin embargo, estos trabajos tienen también mucho interés para esta tesis, ya que aunque no haya tiempos de setup, los fundamentos teóricos de la metaheurística son los mismos.

En [140] Taillard introduce el primero de dichos métodos de búsqueda aplicado al problema job shop. Se compara con un algoritmo shifting bottleneck y con uno de enfriamiento simulado y obtiene en general mejores resultados, mejorando además las mejores soluciones conocidas en algunas instancias de benchmarks típicos. También detalla una versión para procesamiento en paralelo que es capaz de obtener buenas soluciones en instancias grandes en un tiempo computacional reducido.

En [46] Dell' Amico y Trubian aplican la búsqueda tabú a la minimización del makespan en el JSP. El trabajo explica con detalle varias estructuras de vecindad ya clásicas para el JSP, como son *N1*, *N2*, *NA*, *RNA*, *NB* y *NC*, y se discuten sus propiedades de conectividad y factibilidad. En su algoritmo de búsqueda tabú, generan la solución inicial mediante un método basado en una regla de prioridad. La selección de los vecinos se realiza en base a estimaciones del makespan. En la lista tabú se almacenan todos los arcos invertidos, y un movimiento se considera tabú si como mínimo uno de los arcos que hay que invertir se encuentra en la lista tabú. Optan por un tamaño dinámico de la lista tabú para conseguir un equilibro entre diversificación e intensificación en la búsqueda, y utilizan un mecanismo simple para prevenir ciclos en la búsqueda. Además, si transcurre un cierto número de iteraciones sin mejora, el algoritmo vuelve a recomenzar desde la mejor solución encontrada hasta el momento. Realizan un estudio experimental sobre un conjunto de 53 instancias y obtienen muy buenos resultados.

Nowicki y Smutnicki en [105] proponen un algoritmo de búsqueda tabú muy eficiente, llamado TSAB, para tratar la minimización del makespan en el JSP clásico. Al igual que en

el trabajo de Dell' Amico y Trubian en [46], la solución inicial se genera mediante un método basado en reglas de prioridad, aunque el método es diferente, y en la lista tabú también se introducen los arcos invertidos en cada movimiento. La principal novedad es la utilización de una lista L de soluciones elite, que es una lista de tipo LIFO (Last In First Out) en la que se van guardando todas las soluciones encontradas hasta el momento en la búsqueda en las que se mejora el mejor makespan encontrado hasta el momento. Al cabo de un determinado número de iteraciones sin mejora se extrae el primer elemento de esa lista y se reinicia la búsqueda desde esa solución, pero evitando la posibilidad de repetir el mismo movimiento que se realizó la anterior vez desde esa solución. Otras diferencias incluyen la estructura de vecindad utilizada, que es la denominada N_1 por Mattfeld en [88], o el mecanismo de gestión de ciclos, que en lugar de evitar movimientos que pueden dar lugar a un ciclo, optan por detectar cuándo se produce realmente un ciclo, y al detectarlo reiniciar la búsqueda desde una solución extraída de la lista L. El algoritmo finaliza cuando se intenta extraer una solución de L pero la lista esta vacía.

En [106] Nowicki y Smutnicki utilizan de nuevo su algoritmo TSAB pero con una gran cantidad de mejoras, y lo denominan i-TSAB. Una de las mejoras es la evaluación de vecinos, para la que detallan y enuncian las propiedades de un algoritmo que es capaz de calcular el makespan exacto de cada uno de ellos con un coste computacional relativamente reducido. La complejidad del algoritmo estándar de generación de planificaciones semiactivas es de orden $O(N * M)$, mientras que el propuesto en este trabajo es de orden $O(\text{máx}\{\sum_{n=1}^{N} \log m_n, \sum_{m=1}^{M} \log t_m\})$, siendo m_n el número de operaciones del trabajo n, t_m el número de operaciones que utilizan el recurso m, N el número total de trabajos de la instancia, y M el número total de recursos de la instancia. Esta complejidad es claramente menor que $O(N*M)$, ya que debemos observar que $N*M = \sum_{n=1}^{N} m_n = \sum_{m=1}^{M} t_m$. Otra de las diferencias con TSAB es la utilización de un algoritmo de path relinking cuando finalizan una búsqueda tabú. Optan por recomenzar otra búsqueda tabú desde una solución que esté "a la mitad del camino" entre dos de las mejores soluciones ya obtenidas en anteriores iteraciones. También han mejorado su algoritmo de generación de soluciones iniciales para hacerlo menos complejo computacionalmente, utilizan otro mecanismo adicional de detección de ciclos, y han optimizado sus métodos de cálculo de cabezas, colas y ordenaciones topológicas. El trabajo ofrece resultados muy competitivos en un tiempo de ejecución bastante reducido, y es uno de los mejores algoritmos conocidos para el JSP.

En [64] Grabowsky y Wodecki tratan también la minimización del makespan en el proble-

ma JSP clásico. Presentan varias propiedades nuevas sobre los bloques críticos y las utilizan para desarrollar un algoritmo de búsqueda tabú. Por ejemplo, demuestran que dada una planificación, para obtener otra planificación con un makespan menor, al menos una de las operaciones en un bloque crítico debe moverse antes de la primera operación de su bloque o después de la última operación de su bloque. En consecuencia, su estructura de vecindad consiste en mover una operación de un bloque crítico al inicio o al final de dicho bloque. Comprueban la factibilidad de cada vecino generado, y utilizan además cotas inferiores para evaluarlos. El algoritmo que proponen ofrece muy buenos resultados en un tiempo computacional reducido.

En [162] Zhang et al. proponen una búsqueda tabú combinada con un método de enfriamiento simulado. El enfriamiento simulado se aplica únicamente al introducir elementos en una lista soluciones encontradas por la búsqueda tabú, para poder recurrir a ellas posteriormente. Por lo tanto con este método se puede introducir en la lista de soluciones elite, con una cierta probabilidad, una solución que no sea mejor que el mejor makespan encontrado hasta el momento. Realizan un estudio experimental muy extenso sobre varios bancos de ejemplos y obtienen resultados muy competitivos, consiguiendo las mejores soluciones conocidas para algunas de las instancias de JSP más grandes que se pueden encontrar en los benchmarks más representativos. Este algoritmo es capaz de superar a la mayoría de métodos anteriores, con la posible excepción de i-TSAB.

Otro trabajo importante sobre el JSP que podemos citar es [156], en el que Watson et al. estudian en profundidad el algoritmo i-TSAB y tratan de averiguar cuál es la causa de sus buenos resultados. Para ello comparan una búsqueda tabú básica con otras dos estrategias de búsqueda local, llegando a la conclusión de que la búsqueda tabú no es inherentemente superior a las demás. Posteriormente añaden métodos de diversificación e intensificación en la búsqueda, comparables a los utilizados en i-TSAB, y demuestran que es la correcta combinación de todas las técnicas lo que proporciona los mejores resultados. Otro ejemplo lo tenemos en [161], en donde Zhang et al. intentan extender la vecindad propuesta por Balas y Vazacopoulos en [21] para utilizarla en un método de búsqueda tabú.

En cuanto a la aplicación de la búsqueda tabú a otros problemas distintos de la minimización del makespan en el JSP clásico, podemos citar los siguientes:

Bilge et al. en [28] proponen un algoritmo de búsqueda tabú para la minimización del total tardiness en un problema con máquinas paralelas y tiempos de setup dependientes de la secuencia, y presentan las mejores soluciones para un conjunto de instancias generadas por

Serifoglu y Ulusoy en [124]. La solución inicial del algoritmo de búsqueda tabú se obtiene a través de la regla EDD (Earliest Due Date). La estructura de vecindad esta formada por soluciones obtenidas mediante movimientos que involucran intercambios de un trabajo en una máquina con un trabajo en otra máquina distinta, e inserciones de un trabajo en una máquina a otra máquina, o a otra posición de la misma máquina. Implementan una estrategia que utiliza diferentes valores de *tabu tenure*, o número de iteraciones que una variable mantiene la condición de tabú. Aplican secuencialmente valores pequeños medianos y grandes, y con esto consiguen un equilibrio entre diversificación e intensificación durante la búsqueda tabú a corto plazo. A más largo plazo, la diversificación se activa tras un cierto número de iteraciones sin conseguir mejora, y consiste en un gran incremento en el *tabu tenure* actual. En cuanto a la intensificación a más largo plazo, optan por almacenar un pequeño conjunto de soluciones elite y tras un cierto número de iteraciones sin conseguir mejora, se borra la memoria y la búsqueda tabú se reinicia desde la mejor solución elite.

En [45] De Bontridder propone una búsqueda tabú para minimizar el weighted tardiness en una generalización del job shop que incluye tiempos mínimos de inicio para los trabajos, lapsos de tiempo positivos entre tareas, y un grafo de precedencias generalizado. Utiliza una vecindad basada en la inversión de arcos críticos situados en el borde de un bloque crítico. En el estudio experimental utiliza instancias del job shop clásico y se compara con el método shifting bottleneck propuesto por Singer y Pinedo en [133] y con el método de búsqueda local propuesto por Kreipl en [79], obteniendo resultados competitivos.

Stecco y Cordeau proponen en [135] un método de búsqueda tabú para la minimización del makespan en un problema de una única máquina con tiempos de setup de dos tipos, que dependen del tiempo y de la secuencia. Comparan su método con un algoritmo de ramificación y poda, obteniendo mejores resultados en un tiempo de ejecución más reducido. En [81] Laguna et al. también estudian el problema de una única máquina, con el objetivo de minimizar la suma de los costes de setup y las penalizaciones por retraso.

También es posible combinar la búsqueda tabú con otras metaheurísticas. Por ejemplo, el método propuesto por Nasiri y Kianfar en [102] consiste en un algoritmo *scatter search* (búsqueda dispersa) combinado con búsqueda tabú y un mecanismo de path relinking, y lo utilizan para minimizar el makespan en el problema job shop parcial, que es un caso especial del problema general shop. En [112] Pezzella y Merelli combinan un algoritmo de búsqueda tabú con un método shifting bottleneck y lo denominan TSSB. Huang y Liao en [70], y Eswaramurthy y Tamilarasi en [50] presentan algoritmos híbridos que combinan la

búsqueda tabú con una optimización basada en colonias de hormigas. Otro ejemplo reciente de hibridación, que ya comentamos en el capítulo 4, lo tenemos en [26], en donde Beck et al. combinan el algoritmo i-TSAB propuesto por Nowicki y Smutnicki en [106] con el método *Solution-Guided Multi-Point Constructive Search* propuesto por Beck en [25]. Esta metaheurística híbrida es muy competitiva con los mejores métodos del estado del arte, y en el trabajo comprueban que la combinación de las dos técnicas ofrece mejores resultados que cada una de ellas por separado.

6.4.3. Estructura general de una búsqueda tabú

El algoritmo 6.2 muestra la estructura básica de la búsqueda tabú utilizada en esta tesis, que es similar a otros algoritmos de búsqueda tabú que se describen en la literatura, como por ejemplo en el trabajo de Glover y Laguna [59], el de Nowicki y Smutnicki [105] o el de Dell' Amico y Trubian [46]. En el primer paso se genera y evalúa la solución inicial. Después se realizan una serie de iteraciones, en cada una de ellas se construye la vecindad de la solución actual y se elige uno de esos vecinos para la siguiente iteración. Tras un cierto número de iteraciones sin mejora, la búsqueda se reinicia desde una solución anterior que se extrae de una lista de soluciones elite. La búsqueda tabú finaliza tras un número total de iteraciones *maxGlobalIter* o cuando la lista de soluciones elite se agota, y el algoritmo devuelve la mejor solución encontrada hasta el momento.

6.4.4. La lista tabú

La principal utilidad de la lista tabú es evitar que el proceso de búsqueda vuelva a soluciones ya visitadas en anteriores iteraciones. Para reducir el coste computacional hemos elegido almacenar en la lista tabú atributos de los movimientos, en lugar de atributos de las soluciones. Entonces, la lista tabú guarda los arcos que han sido invertidos recientemente. Estos arcos no pueden ser invertidos de nuevo a no ser que el vecino en cuestión cumpla el criterio de aspiración que explicaremos en la sección 6.4.6. Cuando se elige un vecino para la siguiente iteración, la lista tabú se actualiza con todos los arcos que se invierten para generar dicho vecino. Un vecino se marca como tabú si para llegar a él hay que invertir al menos un arco con la condición de tabú.

El número de iteraciones que un arco se mantiene en la lista tabú se suele denominar *tabu tenure*, y es un parámetro que tiene una gran influencia en la eficacia de una búsqueda tabú. Las estrategias para gestionar el tabu tenure pueden ser estáticas, si se mantiene un

Algoritmo 6.2 Algoritmo de búsqueda tabú

Requiere: Una instancia de un problema de planificación P

Produce: Una planificación s para la instancia P

```
Generar una solución inicial s_0;
Hacer la solución actual s = s_0 y la mejor solución encontrada s_B = s;
globalIter = 0, improveIter = 0;
Vaciar la lista tabú, e introducir s en la lista de soluciones elite L (lista LIFO);
mientras globalIter < maxGlobalIter hacer
  si improveIter = maxImproveIter entonces
    si La lista de soluciones elite L esta vacía entonces
      devuelve La solución s_B,
    si no
      Extraer la solución s_L de la lista de soluciones elite L;
      Hacer la solución actual s = s_L;
      Reinicia el contador de iteraciones sin mejora improveIter = 0;
      Vaciar la lista tabú y la estructura de detección de ciclos;
    fin si
  fin si
  globalIter = globalIter+1, improveIter = improveIter+1;
  Generar vecinos de la solución actual s mediante la estructura de vecindad;
  Sea s* el mejor vecino que ni es tabú ni lleva a un ciclo, o bien que cumple el criterio
  de aspiración. Actualizar de forma adecuada la lista tabú y la estructura de detección
  de ciclos, y hacer s = s*;
  si s* es mejor que s_B entonces
    Hacer s_B = s*, improveIter = 0, e introducir s* en la lista de soluciones elite L;
  fin si
fin mientras
devuelve La solución s_B;
```

valor fijo a lo largo de toda la ejecución, o dinámicas, si el valor varía. Diversos estudios, por ejemplo el realizado por Hao et al. en [67], demuestran que una gestión dinámica suele dar mejores resultados que una gestión estática. Nosotros también hemos constatado este hecho de forma experimental en varios experimentos previos.

En esta tesis utilizaremos una gestión dinámica, tal y como proponen Dell' Amico y Trubian en [46]: el tabu tenure se decrementa en una unidad cuando la solución actual es mejor que la solución anterior, y se incrementa en una unidad en caso contrario. En general, el valor se mantiene dentro del intervalo [*min,max*], siendo *min* y *max* parámetros, excepto cuando la solución actual mejora a la mejor solución encontrada hasta el momento, entonces asignamos un tabu tenure de 1. La lógica de esta gestión dinámica es no restringir la búsqueda cuando encontramos zonas prometedoras del espacio de búsqueda, y restringirla sólo a nuevas soluciones cuando estamos en zonas poco prometedoras y queremos intentar llegar a nuevas y mejores zonas.

Para la elección de los parámetros *min* y *max* también nos basamos en el método recomendado por Dell' Amico y Trubian en [46], y se realiza de la siguiente forma. Cada 50 iteraciones se elige el parámetro *min* aleatoriamente entre los valores 2 y $2 + ((N + M)/3)$, mientras que el parámetro *max* se elige aleatoriamente entre los valores $min + 6$ y $min + 6 + ((N + M)/3)$, siendo N el número de trabajos y M el número de máquinas de la instancia. De esta forma ajustamos el tabu tenure al tamaño de la instancia, y además se añade una cierta aleatoriedad a la ejecución, lo cual puede ser beneficioso sobre todo pensando en la combinación con un algoritmo genético, porque quizás aporte más variedad genética a la población.

6.4.5. Detección de ciclos

El procedimiento de detección de ciclos esta basado en el propuesto por Dell' Amico y Trubian en [46] para el JSP. En una solución cíclica, una solución $s(i)$ después de i iteraciones es igual a una solución $s(i + k)$ después de $i + k$ iteraciones, para algún número positivo k, y el arco elegido para ir desde $s(i + k)$ hasta $s(i + k + 1)$ es el mismo que el arco elegido previamente para ir desde $s(i)$ hasta $s(i + 1)$. Para detectar esta situación, se elige un arco representativo para cada movimiento y se le asocia la estimación de la función objetivo del vecino resultante de ese movimiento. Entonces, la estimación de cada vecino se compara con el valor asociado al arco representativo del movimiento correspondiente. Si dichos valores coinciden durante más de *Tcycle* iteraciones consecutivas, se supone que la búsqueda esta en un ciclo, y dicho vecino se descarta a no ser que satisfaga el criterio de aspiración que se explicará en la siguiente sección. Hemos elegido que el arco representativo de un movimiento será el arco invertido "más grande". Por ejemplo, si se crea un vecino moviendo w antes que v en un bloque crítico de la forma $(b'\ v\ b\ w\ b'')$, el arco representativo de ese vecino será el

arco (v, w). En los experimentos se elige un valor de $Tcycle$ igual a 4, ya que empíricamente ha ofrecido resultados ligeramente mejores en varios experimentos previos.

6.4.6. Selección del vecino y criterio de aspiración

La regla de selección elige al vecino con la menor estimación de la función objetivo, descartando a los vecinos tabú y a los que puedan llevar a un ciclo (a no ser que satisfagan el criterio de aspiración). Uno de los problemas de implementar una lista tabú con atributos parciales es que puede ocurrir que se ponga la condición de tabú a una solución no visitada que podría ser interesante visitar. Por tanto el algoritmo utiliza un criterio de aspiración, que permite a cualquier movimiento ser elegido, siempre y cuando su estimación de la función objetivo sea menor que el valor de la función objetivo de la mejor solución encontrada hasta el momento. Si durante la ejecución ocurriera que todos los posibles movimientos son tabú o llevan a un ciclo, y ninguno cumple el criterio de aspiración, el algoritmo simplemente elegirá uno de los vecinos de forma aleatoria.

6.4.7. Gestión de soluciones elite

Con el objetivo de incorporar a la búsqueda un mecanismo de memoria a más largo plazo, utilizamos la estrategia propuesta por Nowicki y Smutnicki en [105] para el algoritmo TSAB, que a su vez se inspira en una estrategia anterior propuesta por Glover en [58]. Dicha estrategia consiste en que cada vez que una nueva solución mejora a la mejor solución encontrada hasta ese momento, esa solución se guarda en una lista L. Entonces, cada $maxImproveIter$ iteraciones sin mejora, la solución actual se reemplaza con una que se extrae de L. Esta técnica se suele utilizar para evitar en cierta medida que el algoritmo se quede atrapado en zonas poco prometedoras del espacio de búsqueda. La longitud de L, decir $Size(L)$ se restringe a un valor no muy grande para evitar almacenar soluciones obtenidas al inicio de la ejecución y que todavía no eran muy buenas. Experimentalmente hemos comprobado que los mejores resultados se obtienen con valores entre 10 y 20. Cuando L se llena, cada nueva solución que se introduce reemplaza a la más antigua.

Cuando se sustituye una solución actual por una solución extraída de L, debemos pensar qué hacer con el estado del algoritmo. Hemos estudiado varias posibilidades, como por ejemplo restaurar el estado de la lista tabú y de la estructura de detección de ciclos que hubiera en el momento de guardar la solución en L, pero finalmente hemos optado por vaciar y reinicializar tanto la lista tabú como la estructura de detección de ciclos. Hemos elegido

esta última opción porque es más simple que la anterior, y los resultados experimentales han sido casi idénticos. De cualquiera de las formas, es imprescindible volver a reiniciar a 0 el contador del número de iteraciones sin mejora.

Por otra parte, en los experimentos hemos elegido $maxImproveIter = maxGlobalIter/Size(L)$ con el objetivo de que la lista de soluciones nunca llegue a agotarse. Es sencillo ver que eligiendo $maxImproveIter$ de esa forma, en cuanto la lista tenga $Size(L)$ soluciones en su interior, nunca llegará a estar vacía antes de que la búsqueda tabú llegue a $maxGlobalIter$ iteraciones. Hemos comprobado experimentalmente que la lista llega a tener $Size(L)$ soluciones en su interior prácticamente siempre, excepto en instancias muy fáciles de resolver.

6.4.8. Combinación con un algoritmo genético

Cuando utilicemos la búsqueda tabú en combinación con un algoritmo genético, aplicaremos la búsqueda tabú a cada uno de los cromosomas del algoritmo genético, y por tanto esas serán las soluciones iniciales.

También hay que tener en cuenta que en este caso el número de iteraciones de la búsqueda tabú será bastante reducido para que el tiempo de computación no sea excesivamente elevado. En los estudios experimentales utilizaremos valores alrededor de 200 iteraciones, y en este tipo de algoritmos híbridos no se utilizará ninguna gestión de soluciones elite, ya que guardar soluciones y volver a ellas no tendría mucho sentido en una ejecución tan corta de una búsqueda tabú.

Además, en este tipo de ejecuciones tan cortas puede tener más sentido utilizar como condición de parada un número de iteraciones máximo sin obtener mejora, en lugar de un número de iteraciones total. Esto es así debido a que en ejecuciones cortas es fácil que la solución todavía siga mejorando al llegar al número máximo de iteraciones totales, y no parece interesante cortar la búsqueda sin haber llegado ni siquiera a un óptimo local.

6.5. Conclusiones

En este capítulo hemos descrito el esquema general de una búsqueda local. Hemos visto que el esquema básico de escalada tiene el grave problema de que se queda atascado en óptimos locales, y exploramos las diferentes formas de resolver este problema. Las tres principales formas son aceptar movimientos que empeoran la solución actual, modificar la

estructura de entornos, o recomenzar la búsqueda desde otro punto. A continuación nos hemos centrado en la búsqueda tabú, realizando una revisión bibliográfica sobre sus aplicaciones a problemas de scheduling, y describiendo los distintos componentes de la búsqueda tabú utilizada en esta tesis.

Capítulo 7

ESTRUCTURAS DE VECINDAD

7.1. Introducción

Los algoritmos de búsqueda local comparten la idea básica de entorno. Dada una solución, una modificación parcial en ella genera una solución de su entorno que difiere ligeramente de la solución origen. Es esperable que la solución del entorno produzca un valor de la función objetivo de calidad similar a su solución origen ya que ambas comparten la mayoría de sus características. Se puede decir que una solución del entorno está en la vecindad de aquella que la origina. Los algoritmos de búsqueda local se concentran en la búsqueda en entornos pues la probabilidad de encontrar una solución mejorada en el entorno es mucho mayor que en zonas menos correlacionadas del espacio de búsqueda.

Los conceptos de camino crítico y bloque crítico son muy importantes en el JSP, ya que la mayoría de propiedades formales, métodos de resolución y estructuras de vecindad propuestas en la literatura se basan en ellos. En general utilizan varios resultados demostrados por Matsuo et al. en [87], por Van Laarhoven et al. en [148], por Nowicki y Smutnicki en [105] o por Grabowsky y Wodecki en [64]. En general, dada una planificación H, consideran sólo un camino crítico de H e intercambian o cambian de posición operaciones en ese camino para conseguir mejoras, tratando de evitar movimientos que no vayan a mejorar inmediatamente el makespan.

En este capítulo describiremos dos de las estructuras de vecindad clásicas para el JSP, una basada en intercambio y otra en inserción, y propondremos su extensión al SDST-JSP,

estudiando las propiedades de factibilidad de los individuos generados, posibles condiciones de no mejora que permitan descartar rápidamente algunos de los vecinos, y algoritmos de estimación de la función objetivo para poder elegir uno de ellos de forma rápida. Describiremos también la forma de calcular la función objetivo exacta del vecino elegido finalmente de la forma más eficiente posible.

Comenzaremos describiendo las características que debería tener una estructura de vecindad eficiente. A continuación propondremos una estructura que denominaremos N_1^S que considerará inversiones de un único arco crítico, y que será una extensión al SDST-JSP de la conocida estructura del JSP denominada N_1 por Mattfeld en [88]. Más adelante propondremos otra estructura de vecindad denominada N^S y que considera la inserción de una tarea crítica en otra posición, lo que supone implícitamente inversiones en el orden de procesamiento de varias tareas críticas en el mismo movimiento. Analizaremos todos los posibles movimientos en ambas estructuras de vecindad y demostraremos la necesidad de establecer condiciones que garanticen la factibilidad de las soluciones vecinas. Propondremos también condiciones suficientes de no mejora que permiten descartar vecinos que se puede saber a priori que no mejorarán la función objetivo de forma inmediata. Después trataremos el tema del número de caminos críticos que consideraremos para crear vecinos de una determinada planificación. Por último, propondremos algoritmos de estimación para las diferentes funciones objetivo estudiadas en esta tesis, que nos permitirán elegir uno de los vecinos de forma rápida, sin necesidad de evaluarlos todos de forma exacta. El estudio realizado en este capítulo nos facilitará el diseño de algoritmos eficientes de búsqueda local en presencia de tiempos de setup, y constituye de hecho una de las principales aportaciones de esta tesis.

7.2. Características deseables de una estructura de vecindad

El éxito de la búsqueda local depende de las propiedades de la definición de vecindad utilizada. Algunas de las características deseables de las estructuras de vecindad son las siguientes.

- *Correlación:* es importante que una solución vecina difiera poco de la original. Esta propiedad es la encargada de que la exploración del espacio de búsqueda sea exhaustiva.
- *Mejora:* un movimiento debe tener una probabilidad alta de generar soluciones con

mejor valor de la función objetivo que la solución actual, y para esto último puede ser necesario incorporar en la definición de la vecindad conocimiento específico del problema.

- *Factibilidad:* es deseable que los vecinos generados sean soluciones factibles, para evitar complejos algoritmos de comprobación o reparación.
- *Conectividad:* debería ser posible llegar desde cualquier individuo a la solución óptima global aplicando un número finito de movimientos.
- *Tamaño:* sobre el número de vecinos que se deben generar a partir de una solución dada, hay que tener en cuenta que si se genera un número muy pequeño de vecinos se puede detener el proceso de búsqueda en etapas tempranas. Por el contrario, un gran número de vecinos permitiría al algoritmo aumentar sus posibilidades de mejora, pero puede ser computacionalmente prohibitivo si el coste de evaluación de la función objetivo es alto.

Hay que tener en cuenta que algunas de estas características pueden contradecirse, y el hecho de mejorar una de ellas puede empeorar otras. De todas formas, todas ellas son características deseables que se pueden utilizar para el desarrollo de estructuras de vecindad eficientes.

7.3. La estructura de vecindad N_1^S

En esta sección extenderemos al SDST-JSP una de las estructuras de vecindad más utilizadas en el JSP clásico: la denominada N_1 por Mattfeld en [88]. Para empezar la sección describiremos dicha estructura, y antes de proponer su extensión al SDST-JSP comentaremos varias propiedades típicas del JSP clásico que dejan de cumplirse en el SDST-JSP. Posteriormente propondremos la estructura N_1^S, que al igual que N_1 se basa en invertir un único arco crítico en cada vecino, y describiremos sus condiciones de factibilidad y de no mejora.

7.3.1. La estructura de vecindad N_1 para el JSP

Algunos de los primeros trabajos sobre estructuras de vecindad para el JSP son el de Matsuo et al. en [87] o el de Van Laarhoven et al. en [148]. En estos trabajos se demuestran algunos resultados interesantes, como que invertir un único arco crítico siempre produce

una planificación factible, o que invertir un único arco puede producir una mejora en el makespan si y sólo si el arco invertido esta o bien al principio o bien al final de un bloque crítico. Además, para el JSP la optimalidad de una planificación se puede demostrar si uno de sus caminos críticos esta formado exclusivamente por un bloque crítico, o bien por bloques críticos de tamaño 1; es decir, si en un camino crítico todas las tareas pertenecen a la misma máquina o al mismo trabajo. Con estos resultados se propusieron un cierto número de estructuras de vecindad que dieron lugar a algunos de los mejores métodos para resolver el JSP. Entre ellos se pueden citar los métodos de Dell' Amico y Trubian en [46], de Nowicki y Smutnicki en [105] o en [106], de Balas y Vazacopoulos en [21] o de Zhang et al. en [162]. Además, se han estudiado y demostrado varias propiedades para algunas de las estructuras de vecindad utilizadas. Por ejemplo, la estructura propuesta por Van Laarhoven et al. en [148] y algunas de las definidas por Dell' Amico y Trubian en [46] verifican la propiedad de conectividad. Si esta propiedad se cumple, la estructura de vecindad se puede también utilizar en el esquema de ramificación de un algoritmo exacto de ramificación y poda. Además, para algunas estructuras, si el conjunto de vecinos es vacío para una cierta planificación, entonces dicha planificación es óptima. Éste es el caso de la estructura propuesta por Nowicki y Smutnicki en [105], y en dicho trabajo esta propiedad permite detener la búsqueda en aproximadamente el 20 % de las instancias resueltas en su estudio experimental. En [64] Grabowsky y Wodecki demuestran que en el JSP clásico, para conseguir una planificación vecina con un makespan más bajo que la planificación original, al menos una de las operaciones en un bloque crítico debe moverse antes de la primera operación de su bloque o después de la última operación de su bloque.

Consideraremos primero la estructura de vecindad para el JSP que Mattfeld denomina N_1 en [88]. Esta estructura se basa en la inversión de arcos pertenecientes a bloques críticos, al igual que la vecindad propuesta por Van Laarhoven et al. en [148], pero tratando de evitar movimientos para los que es posible demostrar, a priori, que no van a producir ninguna mejora inmediata en el valor de C_{max}. Las siguientes proposiciones establecen la base sobre la que se asienta esta estructura. Su demostración puede encontrarse en [88].

Proposición 7.3.1. *Sea H una planificación y (v, w) un arco que no esta en un bloque crítico. Entonces, invertir el arco (v, w) no produce ninguna mejora, incluso aunque la planificación resultante sea factible.*

Proposición 7.3.2. *Sea H una planificación y (v, w) un arco que esta en un bloque crítico. Entonces, la planificación H' obtenida a partir de H invirtiendo el arco (v, w) es factible.*

Proposición 7.3.3. *Sea H una planificación y (v, w) un arco que esta en un bloque crítico tal que v no sea la primera operación del bloque crítico ni tampoco w la última operación del bloque crítico. Entonces, invertir el arco (v, w) no produce ninguna mejora.*

Proposición 7.3.4. *Sea H una planificación y (v, w) un arco que esta en un bloque crítico de tamaño mayor que 2 tal que v es la primera operación del primer bloque crítico o w es la última operación del último bloque crítico. Entonces, invertir el arco (v, w) no produce ninguna mejora.*

Entonces, esta estructura considera un conjunto de movimientos denominados "intercambio en el borde de los bloques críticos en un único camino crítico", lo que significa intercambiar parejas de operaciones sólo al comienzo o al final de un bloque crítico. En [88] Mattfeld define las reglas de transformación de N_1 de la siguiente forma:

Definición 7.1 (N_1)**.** *Sea (v, w) el arco formado por las dos primeras o las dos últimas operaciones de un bloque crítico B. Invertir el arco (v, w) se considera una solución vecina. Para el primer bloque crítico del camino crítico sólo se considera (v, w) al final del bloque, mientras que para el último bloque crítico del camino crítico sólo se considera (v, w) al principio del bloque.*

La vecindad N_1 es extremadamente pequeña y produce sólo ligeras perturbaciones. Lleva a soluciones mejores con una probabilidad relativamente alta, y garantiza factibilidad, pero sin embargo no verifica la propiedad de conectividad (ver contraejemplo en [46]).

7.3.2. Primeras propiedades que no se cumplen en el SDST-JSP

Antes de extender N_1 al SDST-JSP, debemos tener en cuenta que la estructura del problema cambia significativamente respecto al JSP tradicional debido a los tiempos de setup, y por ello se requieren nuevas técnicas. Hay pocos resultados en la literatura sobre estructuras de vecindad en el SDST-JSP, por ejemplo en [166] Zoghby et al. prueban el siguiente resultado:

Proposición 7.3.5. *Sea H una planificación para una instancia del SDST-JSP y (v, w) un arco que esta en un bloque crítico. Entonces, la planificación H' obtenida a partir de H invirtiendo el arco (v, w) puede no ser factible.*

Ilustraremos también este hecho con el ejemplo 7.2 de la sección 7.3.3.

Por otra parte, en el JSP tradicional la inversión de un arco crítico que no esté situado en el extremo de un bloque crítico no puede producir mejora, como hemos visto en la proposición 7.3.3. La proposición 7.3.6 demuestra que no ocurre lo mismo en el SDST-JSP. Esto muestra una vez más que el añadido de los tiempos de setup hace el problema más complejo.

Proposición 7.3.6. *Sea H una planificación para una instancia del SDST-JSP y (v, w) un arco que esta en un bloque crítico tal que v no sea la primera operación del bloque crítico ni tampoco w la última operación del bloque crítico. Entonces, invertir el arco (v, w) puede producir mejora.*

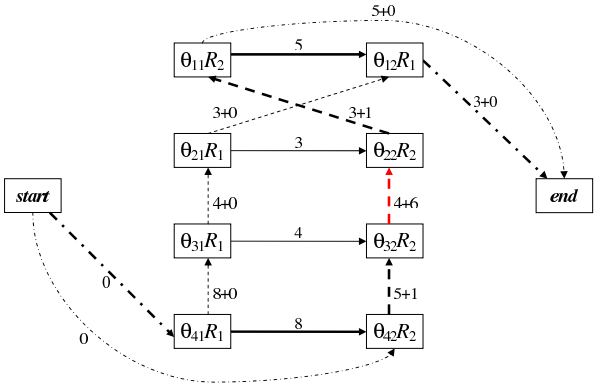
Demostración. El siguiente ejemplo ilustra este hecho. Consideremos la planificación de la instancia de SDST-JSP de la figura 7.1 (a). Es decir, una instancia con cuatro trabajos y dos recursos o máquinas. Las tareas θ_{12}, θ_{21}, θ_{31} y θ_{41} comparten el recurso 1 y dicho recurso no tiene tiempos de setup. Las tareas θ_{11}, θ_{22}, θ_{32} y θ_{42} comparten el recurso 2 y los tiempos de setup no nulos de dicho recurso que son relevantes para este ejemplo son los siguientes: $S_{\theta_{42}\theta_{32}} = 1$, $S_{\theta_{32}\theta_{22}} = 6$, $S_{\theta_{22}\theta_{11}} = 1$, $S_{\theta_{42}\theta_{22}} = 1$, $S_{\theta_{22}\theta_{32}} = 2$, $S_{\theta_{32}\theta_{11}} = 1$. En la figura se marca en negrita el camino crítico, cuya longitud, es decir el makespan, es 36. Esto también se puede apreciar en el diagrama de Gantt de la figura 7.1 (b).

Podemos observar que el camino crítico tiene un bloque crítico de tamaño 4, formado por las tareas θ_{42} θ_{32} θ_{22} y θ_{11}. Las figuras 7.1 (c) y (d) muestran el grafo disyuntivo y el diagrama de Gantt, respectivamente, resultantes de invertir el arco interior de dicho bloque crítico, es decir el arco $(\theta_{32}\theta_{22})$. Hemos marcado en rojo el arco crítico invertido. El makespan de esta nueva planificación es de 33, menor que el de la planificación anterior. □

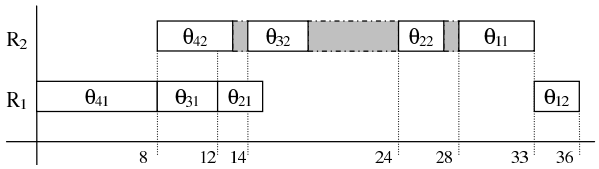
7.3.3. Conectividad y Optimalidad para estructuras de vecindad

La conectividad es una propiedad importante para cualquier estructura de entorno utilizada en una búsqueda local. Asegura que no existen puntos de partida desde los cuales la búsqueda local no pueda alcanzar un óptimo, lo que permitiría diseñar métodos exactos de búsqueda. Sin embargo, el siguiente teorema establece dos nuevos resultados sobre conectividad y optimalidad para estructuras de vecindad en el SDST-JSP que anulan toda esperanza de utilizar este tipo de estructuras de entornos en métodos exactos.

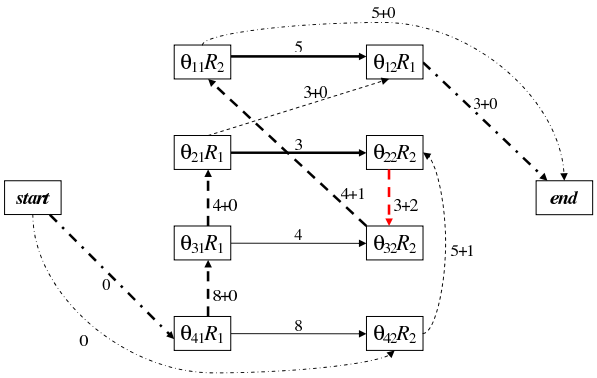
Teorema 7.1. *Sea N una estructura de vecindad para el SDST-JSP tal que $N(H)$ esta formado por planificaciones factibles obtenidas a partir de H invirtiendo exclusivamente*



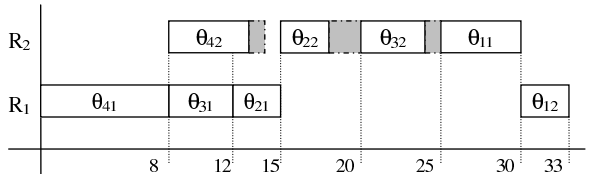
(a) Grafo disyuntivo de la planificación original



(b) Diagrama de Gantt de la planificación original



(c) Grafo disyuntivo de la planificación vecina

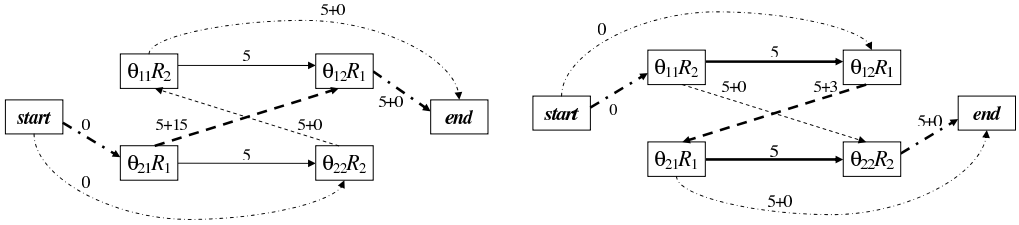


(d) Diagrama de Gantt de la planificación vecina

Figura 7.1: En el SDST-JSP la inversión de un arco perteneciente al interior de un bloque crítico puede producir mejora

órdenes de procesamiento de operaciones críticas. Entonces N no cumple la propiedad de conectividad. Además, H puede no ser óptimo incluso aunque $N(H) = \emptyset$.

Demostración. Estos resultados se pueden probar mediante el ejemplo de la figura 7.2. Sea H la planificación de la figura (a). Como se puede observar, el único bloque crítico de H es el dado por el arco $(3, 2)$, e invertir dicho arco da lugar a un ciclo en el grafo y por tanto a una planificación no factible, por lo tanto $N(H) = \emptyset$. Sin embargo, H no es óptima, ya que su makespan es mayor que el de la planificación de la figura (b), que es la única solución óptima para este problema. Para alcanzar la solución óptima a partir de H, habría que invertir simultáneamente un arco crítico y un arco no crítico.



(a) Grafo sin vecinos

(b) Solución óptima para el mismo problema

Figura 7.2: Contraejemplo a la conectividad de estructuras de entorno basadas en la inversión de arcos críticos

□

El hecho de no poder garantizar la conectividad es claramente un problema, pero no suele tener excesiva relevancia en el contexto de un algoritmo de búsqueda local en donde el número máximo de iteraciones es limitado, o el criterio de aceptación esta basado en la escalada. Sin embargo esto muestra otra dificultad añadida del SDST-JSP respecto al JSP.

7.3.4. Condiciones de factibilidad

En el caso del JSP sin tiempos de setup la factibilidad siempre se mantiene en el caso de invertir un único arco crítico. Sin embargo en el SDST-JSP sólo se mantendrá si no existe ningún camino alternativo entre las operaciones v y w antes de invertir el arco crítico (v, w). Esta situación se muestra en la figura 7.3. Observemos que en el JSP no se puede producir esta situación, ya que si existiera un camino alternativo tendría que ser un camino más largo, y por tanto el arco (v, w) no podría pertenecer al camino crítico. El siguiente resultado proporciona una condición muy rápida de evaluar para que no exista ningún camino

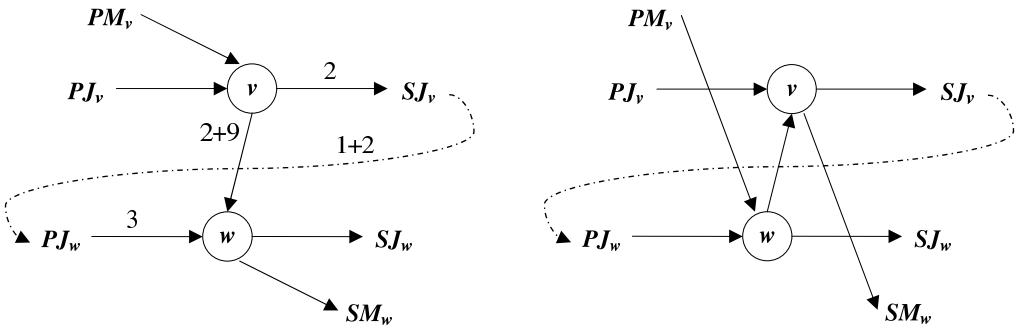


Figura 7.3: Inversión de un arco en un bloque crítico con un camino alternativo

alternativo. Sin embargo, debido a que esta condición es suficiente, pero no necesaria, se pueden descartar algunas planificaciones factibles.

Proposición 7.3.7. *Sea H una planificación y (v, w) un arco de un bloque crítico. Una condición suficiente para que no exista un camino alternativo entre v y w es que*

$$r_{PJ_w} < r_{SJ_v} + p_{SJ_v} + \min\{S_{kl} | (k,l) \in E, J_k = J_v\} \tag{7.1}$$

siendo J_k el trabajo al que pertenece la operación k.

Demostración. Vamos a suponer que se verifica la condición 7.1 y que existe un camino alternativo C entre v y w, entonces el sucesor de v en este camino es SJ_v y el predecesor de w es PJ_w (ver figura 7.3). Sea M_u la máquina requerida por la tarea u, y sea C' la parte de C desde SJ_v hasta PJ_w, entonces tenemos que

$$\begin{aligned} r_{PJ_w} &\geq r_{SJ_v} + cost(C') \\ &= r_{SJ_v} + \sum(p_k | (k,l) \in C') + \sum(S_{kl} | (k,l) \in C', M_k = M_l) \\ &\geq r_{SJ_v} + p_{SJ_v} + \sum(S_{kl} | (k,l) \in C', M_k = M_l) \end{aligned}$$

Como existe un arco $(k,l) \in C'$ con $J_k = J_{SJ_v} = J_v$ y $M_k = M_l$ (es decir, $(k,l) \in E$), entonces

$$r_{PJ_w} \geq r_{SJ_v} + p_{SJ_v} + \min\{S_{kl} | (k,l) \in E, J_k = J_v\}$$

Lo cual es una contradicción con la condición 7.1 presentada anteriormente. □

Los resultados sobre factibilidad presentados en esta sección se pueden aplicar a todas las funciones objetivo estudiadas en esta tesis: makespan, maximum lateness, weighted tardiness y total flow time. Podemos observar que las condiciones para que los grafos solución

contengan un ciclo son equivalentes independientemente de la función objetivo que desee minimizar. Por otra parte, las demostraciones realizadas sólo utilizan información sobre las cabezas de las tareas, sus duraciones y los tiempos de setup, y todos estos datos se definen de la misma forma para esas funciones objetivo.

7.3.5. Condiciones de no mejora

La idea principal de N_1 para el JSP, tal y como explican Nowicki y Smutnicki en [105], es considerar, de todos los movimientos simples en una planificación (inversiones de un único arco crítico), sólo aquellos que puedan llevar a una mejora inmediata del makespan. Ya comentamos que en el JSP clásico la inversión de un arco (v, w) que esta en el interior de un bloque crítico, es decir, tal que PM_v y SM_w pertenezcan también al mismo bloque crítico, no puede producir mejora. Por esta razón para definir N_1 se reducen los movimientos de la estructura de vecindad definida por Van Laarhoven et al. en [148] a los bordes de los bloques críticos. Sin embargo, para extender esta estructura al SDST-JSP también consideraremos los movimientos en el interior de un bloque crítico porque en este contexto, como ya hemos ilustrado, sí pueden producir mejora.

A continuación veremos como, al igual que en el JSP clásico, la inversión de un arco no crítico no puede producir mejora.

Proposición 7.3.8. *Sea H una planificación y (v, w) un arco disyuntivo que no pertenece a ningún bloque crítico. Entonces, invertir el arco (v, w) no puede producir mejora incluso aunque la planificación resultante sea factible.*

Demostración. Si el arco (v, w) no pertenece a ningún bloque crítico, (v, w) no pertenece a ningún camino crítico de H. Sea P un camino crítico de H. Entonces, P también es un camino en la planificación H' resultante de invertir (v, w) en H. Entonces, si la planificación resultante H' es factible, $C_{max}(H') \geq cost(P) = C_{max}(H)$. □

La proposición 7.3.8 es cierta para cualquiera de las funciones objetivo estudiadas en esta tesis. Sin embargo, en el caso del weighted tardiness y el total flow time debemos considerar en la demostración al menos un camino crítico hacia cada uno de los diferentes nodos finales, en lugar de un único camino P.

A continuación estableceremos condiciones suficientes para la no mejora cuando se invierte un arco del interior de un bloque crítico o de los extremos del camino crítico en una solución H del problema.

Proposición 7.3.9. *Sea H una planificación y (v, w) un arco en el interior de un bloque crítico B, es decir, PM_v y SM_w pertenecen a B. Incluso aunque la planificación H', obtenida a partir de H invirtiendo el arco (v, w), sea factible, H' no mejora a H si la siguiente condición se cumple*

$$S_{xw} + S_{wv} + S_{vy} \geq S_{xv} + S_{vw} + S_{wy}. \tag{7.2}$$

donde $x = PM_v$ y $y = SM_w$ en la planificación H.

Demostración. Para todo nodo v de H, $r_v + p_v + q_v$ es la longitud del camino más largo desde el nodo *start* hasta el nodo *end* a través del nodo v. Si v pertenece a un camino crítico, este valor es el makespan de H, y si no, es una cota inferior del makespan.

Como (v, w) es un arco en el interior de un bloque crítico, en el camino crítico existirá una secuencia de operaciones que requieren la misma máquina, de la forma $(x\ v\ w\ y)$ con $C_{max} = r_y + p_y + q_y$

Sea H' el vecino obtenido a partir de H invirtiendo el arco (v, w). En H' la secuencia de operaciones $(x\ v\ w\ y)$ se procesan en el orden $(x\ w\ v\ y)$, mientras que para todas las operaciones anteriores a x y todas las posteriores a y en el camino crítico de H, los órdenes de procesamiento son idénticos en H' y en H. Entonces, $r'_x = r_x$, $q'_y = q_y$

Pero la secuencia $(x\ w\ v\ y)$ puede no pertenecer a ningún camino crítico de H', por lo tanto,

$$C'_{max} \geq r'_y + p_y + q'_y$$

Ya que $q'_y = q_y$, $C'_{max} \geq C_{max}$ se cumple si $r'_y \geq r_y$. Ahora, hay que tener en cuenta que en H

$$r_y = r_x + p_x + S_{xv} + p_v + S_{vw} + p_w + S_{wy}$$

Mientras que en H'

$$\begin{aligned} r'_y &\geq r'_v + p_v + S_{vy} \\ &\geq r'_w + p_w + S_{wv} + p_v + S_{vy} \\ &\geq r'_x + p_x + S_{xw} + p_w + S_{wv} + p_v + S_{vy} \end{aligned}$$

Y como $r'_x = r_x$, entonces

$$r'_y \geq r_x + p_x + S_{xw} + p_w + S_{wv} + p_v + S_{vy}$$

Entonces, si la condición (7.2) se cumple, es decir:

$$S_{xw} + S_{wv} + S_{vy} \geq S_{xv} + S_{vw} + S_{wy}$$

tenemos que $r'_y \geq r_y$ y por tanto $C'_{max} \geq C_{max}$. □

Proposición 7.3.10. *Sea H una planificación y (v, w) un arco de un bloque crítico siendo v la primera operación del primer bloque crítico; y sea y el sucesor de w en el bloque crítico. Incluso aunque invertir el arco (v, w) lleve a una planificación factible, no habrá ninguna mejora si la siguiente condición se cumple*

$$S_{0w} + S_{wv} + S_{vy} \geq S_{0v} + S_{vw} + S_{wy} \tag{7.3}$$

Existe un resultado análogo si w es la última operación del último bloque crítico.

Demostración. Como (v, w) pertenece a un camino crítico y y es el sucesor de w en un bloque crítico, tenemos que

$$r_y = r_v + p_v + S_{vw} + p_w + S_{wy}$$

ya que v es la primera operación del primer bloque crítico, $r_v = S_{0v}$, entonces

$$r_y = S_{0v} + p_v + S_{vw} + p_w + S_{wy}$$

Ahora, sea H' el vecino obtenido invirtiendo el arco (v, w). Entonces, las operaciones v, w e y de H' se procesan en el orden $(w\ v\ y)$, mientras que las operaciones posteriores a y en H se procesan en el mismo orden en H' y en H. Entonces, $q'_y = q_y$, y mediante un razonamiento similar al aplicado en la anterior demostración, tenemos que $C'_{max} \geq r'_y + p_y + q'_y$. Entonces, $C'_{max} \geq C_{max}$ si $r'_y \geq r_y$, y además

$$\begin{aligned} r'_y &\geq r'_v + p_v + S_{vy} \\ &\geq r'_w + p_w + S_{wv} + p_v + S_{vy} \end{aligned}$$

Ya que w es la primera operación que se procesa en su máquina en la planificación H'

$$r'_y \geq S_{0w} + p_w + S_{wv} + p_v + S_{vy}$$

Entonces, si la condición (7.3) se cumple, es decir:

$$S_{0w} + S_{wv} + S_{vy} \geq S_{0v} + S_{vw} + S_{wy}$$

tenemos que $r'_y \geq r_y$ y entonces $C'_{max} \geq C_{max}$. □

Estos resultados establecen condiciones suficientes para la no mejora de todos los posibles movimientos elementales en una planificación H, es decir, movimientos que implican invertir

únicamente un arco crítico. Por supuesto, si (v, w) es el único arco en su bloque crítico, únicamente se considera la inversión de dicho arco.

Se pueden demostrar fácilmente resultados equivalentes para la minimización del maximum lateness, pero por desgracia la generalización de las proposiciones 7.3.9 y 7.3.10 al caso de funciones objetivo de tipo suma no es sencilla, debido a que el grafo disyuntivo tendrá varios caminos críticos hacia varios nodos finales distintos. Por ello, aunque sepamos que una determinada inversión no podrá acortar un camino crítico, sí que podría acortar otros caminos hacia nodos finales distintos, y por tanto la función objetivo total podría ser menor. Además, comprobar que un determinado movimiento no puede acortar ninguno de los caminos críticos en una planificación es mucho más costoso computacionalmente. Esto contradice la premisa de que la condición de no mejora se debería poder evaluar rápidamente, y por ello decidimos no utilizarla en funciones objetivo como el weighted tardiness o el total flow time.

7.3.6. Definición de N_1^S

Los resultados proporcionados en las anteriores secciones nos permiten definir la estrategia de vecindad como sigue:

Definición 7.2. (N_1^S) *Dada una planificación H, la vecindad $N_1^S(H)$ consiste en todas las planificaciones obtenidas a partir de H invirtiendo un arco (v, w) de un bloque crítico del grafo solución asociado a H, siempre y cuando ninguna de las condiciones dadas en las proposiciones* 7.3.9 *y* 7.3.10 *se cumpla, y que la condición de factibilidad dada en la proposición* 7.3.7 *sí que se cumpla.*

Esta vecindad se podrá utilizar en la minimización del makespan y del maximum lateness. Para la minimización del weighted tardiness y del total flow time la vecindad se define de la misma forma, pero sin tener en cuenta las condiciones de no mejora de las proposiciones 7.3.9 y 7.3.10, debido a que no hemos demostrado resultados equivalentes para dichas funciones objetivo.

Por último, para remarcar la aportación que supone la estructura de vecindad N_1^S propuesta aquí, también hemos considerado una extensión de N_1 mucho más trivial, que consiste únicamente en intercambiar las tareas que estén al borde de un bloque crítico, al igual que la estructura N_1 para el JSP. Es decir, no se realiza ningún intercambio de tareas en el interior de un bloque crítico. De todas formas, incluso aunque sólo se hagan cambios en los bordes

de un bloque crítico, en el SDST-JSP se debe comprobar la factibilidad del nuevo vecino, como ya hemos demostrado a lo largo de este capítulo. Varios experimentos han concluído que esta extensión trivial produce resultados mucho peores que N_1^S. En la sección 8.3.3 mostraremos resultados sobre comparaciones experimentales entre estructuras de vecindad en la minimización del makespan, mientras que en la sección 10.3.2 haremos lo propio con el weighted tardiness y en la sección 11.2 con el total flow time.

7.4. La estructura de vecindad N^S

En esta sección propondremos una estructura de vecindad para el SDST-JSP que denominaremos N^S, y que está basada en un movimiento de inserción en lugar de intercambio y que, en consecuencia, puede invertir el orden de procesamiento de varios pares de tareas críticas en un mismo movimiento. Esta estructura esta inspirada en otras estructuras previas para el JSP clásico, especialmente en la denominada *NB* por Dell' Amico y Trubian en [46] o N_2 por Mattfeld en [88]. En esta sección generalizaremos las condiciones de no mejora y de factibilidad desarrolladas para la estructura N_1^S con el objetivo de que también sean útiles para vecinos formados realizando movimientos más amplios dentro de un bloque crítico, y a partir de estos resultados definiremos la estructura N^S.

7.4.1. La estructura de vecindad N_2 para el JSP

A continuación describiremos una de las estructuras de vecindad más utilizadas en el JSP clásico: la estructura *NB* definida por Dell' Amico y Trubian en [46] y denominada N_2 por Mattfeld en [88]. La idea general de la estructura N_2 es considerar movimientos de tipo inserción de cualquier tarea dentro de un bloque crítico a otra posición del bloque. En este sentido, N_2 plantea movimientos en un rango más amplio que N_1. A continuación mostramos la definición de la estructura tal y como se explica en [46].

Definición 7.3 (N_2)**.** *Sea la operación v miembro de un bloque crítico B. En una solución vecina, v se mueve al principio o al final de B, si la planificación resultante es factible. En caso contrario se mueve lo más cerca posible de la primera o de la última operación de B para la cual la factibilidad se conserve.*

Una de las ventajas que ofrece N_2 respecto a la estructura N_1 detallada en la sección 7.3.1 es que cumple la propiedad de conectividad en el JSP, aunque ya hemos explicado en la

sección 7.3.3 que esta propiedad no se cumplirá en el SDST-JSP. Uno de los inconvenientes con respecto a N_1 es que el número de vecinos es mucho mayor, y además muchos de ellos no llevarán a una mejora inmediata del makespan. Por ejemplo, los vecinos que sean consecuencia de intercambios de tareas tales que ninguna está en el extremo de un bloque crítico no producirán una mejora inmediata en el JSP, pero sin embargo en el SDST-JSP sí que podrían producirla bajo ciertas condiciones, tal y como demostraremos en la sección 7.4.3.

También observamos que N_2 puede invertir el orden de procesamiento de varios pares de tareas críticas en un mismo movimiento, por lo que tanto en el JSP como en el SDST-JSP será necesario comprobar la factibilidad del vecino resultante. A continuación, en la sección 7.4.2 describiremos varias alternativas para la comprobación de la factibilidad.

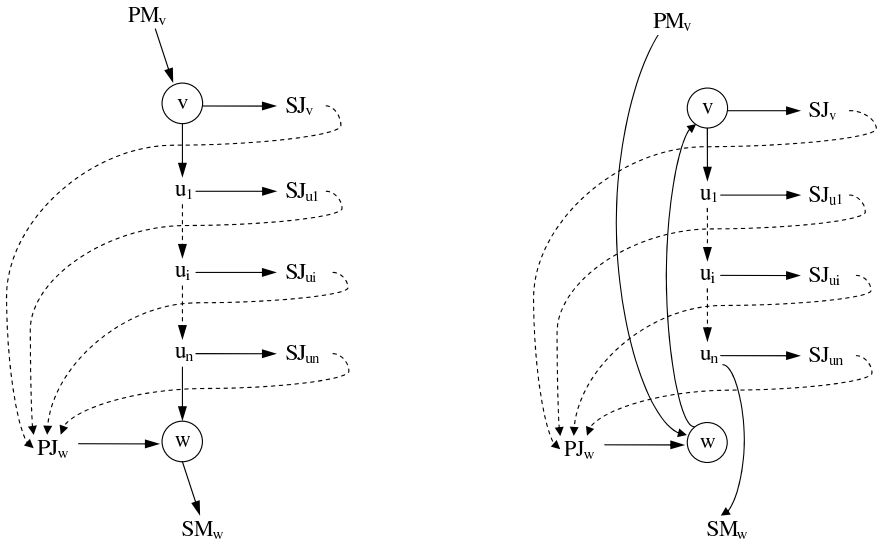
7.4.2. Condiciones de factibilidad

En esta sección generalizaremos las condiciones de factibilidad de la sección 7.3.4 para el caso de varias inversiones de órdenes de procesamiento de pares de tareas críticas en el mismo movimiento. Los dos siguientes resultados explican bajo qué condiciones el nuevo grafo solución contendrá un ciclo en una estructura de vecindad de tipo inserción.

Proposición 7.4.1. *Dado un bloque crítico B de la forma $(b'\ v\ b\ w\ b'')$, donde b, b' y b'' son secuencias de operaciones que pueden ser vacías, se tiene que la secuencia $(b'\ w\ v\ b\ b'')$ no es factible si y sólo si hay un camino alternativo en el grafo solución del individuo original desde u hasta w, para algún $u \in b \cup \{v\}$, y dicho camino alternativo debe pasar necesariamente a través de PJ_w.*

Demostración. Si la operación w se mueve antes que la operación v para tener $(b'\ w\ v\ b\ b'')$, todos los órdenes de procesamiento de los pares de tareas críticas del conjunto $\{(u, w) : u \in b \cup \{v\}\}$ se invierten. Entonces, existirá un ciclo en la nueva solución si y sólo si hay un camino desde u hasta w, para algún $u \in b \cup \{v\}$ en el grafo solución de la solución vecina. Así pues, los caminos alternativos que pueden dar lugar a un ciclo tienen que pasar necesariamente a través de PJ_w, ya que el arco (PM_w, w) no existirá en la nueva planificación. Obviamente esos caminos tienen que existir también en el grafo solución original. □

La figura 7.4 muestra los caminos alternativos que causarían un ciclo en el nuevo grafo si movemos la operación w justo antes de v. En la figura se puede apreciar que los caminos alternativos deben pasar a través de PJ_w, tal y como explicamos en la proposición 7.4.1.



(a) Solución inicial (b) Solución vecina

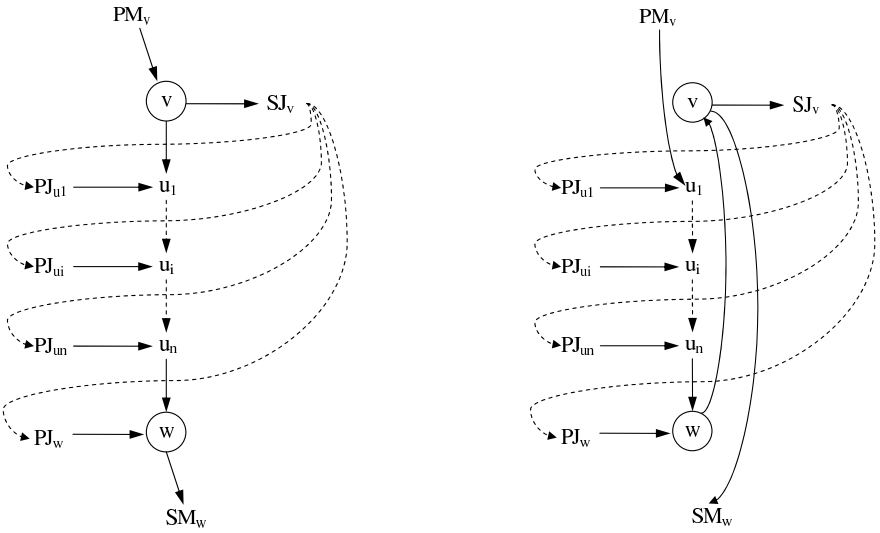
Figura 7.4: Posibles caminos alternativos que pueden existir en un bloque crítico, que darían lugar a un ciclo tras mover la operación w antes de la operación v

A continuación se muestra un resultado análogo para los movimientos hacia el final de un bloque crítico.

Proposición 7.4.2. *Dado un bloque crítico B de la forma $(b'\ v\ b\ w\ b'')$, donde b, b' y b'' son secuencias de operaciones que pueden ser vacías, se tiene que la secuencia $(b'\ b\ w\ v\ b'')$ no es factible si y sólo si hay un camino alternativo en el grafo solución del individuo original desde v hasta u, para algún $u \in b \cup \{w\}$, y dicho camino alternativo debe pasar necesariamente a través de SJ_v.*

Demostración. Si la operación v se mueve después de la operación w para tener $(b'\ b\ w\ v\ b'')$, todos los órdenes de procesamiento de los pares de tareas críticas del conjunto $\{(v,u) : u \in b \cup \{w\}\}$ se invierten. Entonces, existirá un ciclo en la nueva solución si y sólo si hay un camino desde v hasta u, para algún $u \in b \cup \{w\}$ en el grafo solución de la solución vecina. Los caminos alternativos que pueden dar lugar a un ciclo tienen que pasar necesariamente a través de SJ_v, ya que el arco $(v,\ SM_v)$ no existirá en la nueva planificación. Todos esos caminos tienen que existir también en el grafo solución original, por supuesto. □

La figura 7.5 muestra los caminos alternativos que causarían un ciclo en el nuevo grafo si movemos la operación v justo después de la operación w. En la figura se puede apreciar



(a) Solución inicial (b) Solución vecina

Figura 7.5: Posibles caminos alternativos que pueden existir en un bloque crítico, que darían lugar a un ciclo tras mover la operación v después de la operación w

que los caminos alternativos deben pasar a través de SJ_v, tal y como explicamos en la proposición 7.4.2.

Una comprobación exacta de que no existe ninguno de los caminos indicados en las proposiciones 7.4.1 y 7.4.2 se puede realizar con un algoritmo estándar de etiquetado como el propuesto por Kahn en [74]. Pero como este procedimiento es costoso computacionalmente, es habitual establecer alguna condición suficiente pero no necesaria que pueda ser evaluada rápidamente, al igual que ya hicimos en la sección 7.3.4 con las inversiones de un único arco crítico.

La proposición 7.4.3 establece una condición suficiente para garantizar que no existe un camino alternativo desde una operación v hasta otra operación w de su mismo bloque crítico que pase a través de SJ_v y de PJ_w. Recordemos que la cabeza r_v de una operación v es la longitud del camino más largo desde el nodo *start* hasta v en el grafo solución, y denominamos J_k al trabajo al que pertenece la operación k.

Proposición 7.4.3. *Dado un bloque crítico de la forma* $(b'\ v\ b\ w\ b'')$*, donde* b*,* b' *y* b'' *son secuencias de operaciones que pueden ser vacías, una condición suficiente para garantizar que no existe un camino alternativo desde* v *hasta* w *que pase a través de* SJ_v *y de* PJ_w *es*

que

$$r_{PJ_w} < r_{SJ_v} + p_{SJ_v} + \min\{S_{kl} | (k,l) \in E, J_k = J_v\} \tag{7.4}$$

Demostración. La demostración es similar a la de la proposición 7.3.7 mostrada en la sección 7.3.4. □

Esta condición se puede utilizar en cualquier estructura de vecindad consistente en invertir órdenes de procesamiento de tareas en un bloque crítico, y debe ser evaluada para cada uno de los órdenes de procesamiento que se inviertan en el vecino. Finalmente, los dos siguientes corolarios indican la forma de comprobar la factibilidad de un vecino creado mediante un movimiento de tipo inserción.

Corolario 7.4.1. *Dado un bloque crítico de la forma* $(b'\ v\ b\ w\ b'')$*, donde* b*,* b' *y* b'' *son secuencias de operaciones que pueden ser vacías. Una condición suficiente para garantizar la factibilidad de un vecino generado insertando la tarea* w *justo antes de* v*, de la forma* $(b'\ w\ v\ b\ b'')$ *es que*

$$r_{PJ_w} < r_{SJ_u} + p_{SJ_u} + \min\{S_{kl} | (k,l) \in E, J_k = J_u\} \forall u \in \{v\} \cup b \tag{7.5}$$

Demostración. La demostración es inmediata, teniendo en cuenta las proposiciones 7.4.1 y 7.4.3. □

Corolario 7.4.2. *Dado un bloque crítico de la forma* $(b'\ v\ b\ w\ b'')$*, donde* b*,* b' *y* b'' *son secuencias de operaciones que pueden ser vacías. Una condición suficiente para garantizar la factibilidad de un vecino generado insertando la tarea* v *justo antes de* w*, de la forma* $(b'\ b\ w\ v\ b'')$ *es que*

$$r_{PJ_u} < r_{SJ_v} + p_{SJ_v} + \min\{S_{kl} | (k,l) \in E, J_k = J_v\} \forall u \in \{w\} \cup b \tag{7.6}$$

Demostración. La demostración es inmediata, teniendo en cuenta las proposiciones 7.4.2 y 7.4.3. □

Es importante aclarar que estas condiciones se pueden evaluar de forma muy eficiente si las aplicamos eligiendo las operaciones u en un determinado orden. Por ejemplo, dada una operación w, todas las posiciones a las que se puede mover hacia atrás w se pueden determinar con una única iteración sobre las operaciones que preceden a w en el bloque crítico. Es decir, se comprueba la desigualdad de la condición 7.4 posición a posición, y en cuanto se encuentre una tarea para la que no se cumpla, entonces sabemos que la tarea w se

puede mover a cualquier posición entre w y la tarea para la que la condición no se cumplió, pero nunca más lejos. Por lo tanto se pueden determinar todos los posibles vecinos en tiempo polinomial. El mismo razonamiento se puede aplicar a los movimientos de tareas hacia el final de su bloque crítico.

De todas formas, descartar los vecinos que puedan dar lugar a soluciones no factibles no es la única alternativa posible. Por ejemplo, en [97] Murovec y Suhel tratan la minimización del makespan en el job shop clásico mediante una combinación de algoritmo genético y búsqueda local, y en su estructura de vecindad, en lugar de rechazar los vecinos que dan lugar a soluciones no factibles aplican un mecanismo de reparación a dichas soluciones para convertirlas en factibles. Sin embargo, la opción de descartar los vecinos no factibles es, con diferencia, la más ampliamente utilizada en la literatura.

Al igual que ocurría con los resultados ofrecidos para la factibilidad de N_1^S de la sección 7.3.4, los resultados sobre factibilidad presentados para N^S en esta sección se pueden aplicar a todas las funciones objetivo estudiadas en esta tesis. De nuevo, las condiciones para que los grafos contengan un ciclo son equivalentes para todas las funciones objetivo, y las demostraciones realizadas sólo utilizan información sobre las cabezas de las tareas, sus duraciones y los tiempos de setup, es decir datos que se definen de la misma forma para las cuatro funciones objetivo estudiadas.

Cálculo exacto de la factibilidad de los vecinos

Hasta aquí, en esta sección hemos descrito una condición suficiente para determinar la factibilidad de un vecino. A continuación describiremos un método que permitirá evaluar la factibilidad de forma mucho más precisa, o incluso de forma exacta, dependiendo de un parámetro de control introducido a priori. Como es natural, mayor precisión lleva asociada mayor complejidad, y la decisión de asumir este incremento en la carga computacional estará condicionada por los resultados experimentales. Al igual que antes, dado un par ordenado por su procesamiento de tareas v y w del mismo bloque crítico, (v, w), se desea saber si existe un camino alternativo entre v y w que pudiera dar lugar a un ciclo en el grafo resultante de invertir el orden de procesamiento de ambas tareas.

La idea general es recorrer todos los caminos posibles que parten de SJ_v hasta descubrir un camino alternativo hasta w que pase a través de PJ_w, podando además aquellos para los que se sepa que ya no pueden llegar a ser un camino alternativo. Además dispone de un parámetro de control llamado $maxNodos$ con el que podremos controlar el número máximo

de nodos que se visitarán y que nos servirá para reducir el coste computacional a cambio de calcular la factibilidad con una precision menor. El algoritmo continúa explorando caminos hasta que se cumple una de las siguientes condiciones:

1. Se descubre un camino alternativo hasta PJ_w, en cuyo caso se demuestra la no factibilidad.

2. Se han podado todos los caminos posibles sin descubrir un camino alternativo, y por lo tanto se demuestra la factibilidad del vecino.

3. Se ha visitado el número máximo de nodos indicado por el parámetro *maxNodos* sin haber encontrado un camino alternativo. En este caso no se puede demostrar que no exista un camino alternativo, y el resultado del algoritmo es que el vecino es no factible, aunque podría serlo realmente.

El criterio de poda que utiliza el algoritmo es el siguiente: si $r_x + p_x > r_{PJ_w}$ perteneciendo x al camino que se esta revisando, entonces es claro que x no puede pertenecer a un camino alternativo.

Como hemos comentado *maxNodos* indica el número máximo de nodos que visitará el algoritmo, y por lo tanto si elegimos un valor igual o superior al número de tareas del problema, este algoritmo calculará de forma exacta si existe un camino alternativo o no. Por el contrario, cuanto más bajo sea el valor *maxNodos* menos precisa será la estimación, aunque el algoritmo consumirá un tiempo de ejecución menor. Experimentalmente hemos comprobado que, utilizando $maxNodos = 5$, se demuestra la factibilidad del 75 % de los vecinos que la condición de factibilidad simple explicada en la sección 7.4.2 marca como no factibles pero que realmente sí que son factibles.

El algoritmo 7.1 muestra el pseudocódigo del algoritmo. La finalidad del conjunto *nodos_considerados* es almacenar los nodos alcanzados hasta el momento, para evitar evaluar varias veces caminos diferentes que pasan por el mismo nodo.

En la sección 8.3.11 realizamos un estudio experimental para comprobar si calcular la factibilidad de los vecinos con más precisión es una opción mejor que utilizar la comprobación simple ya explicada anteriormente. Sin embargo, los resultados mostrarán que por lo general es más eficiente utilizar la condición simple porque consume un menor tiempo computacional y los resultados son similares. Esto último probablemente es debido a que los vecinos que descarta la condición simple no son prometedores de todas formas, por lo que descartarlos no perjudica el proceso de búsqueda local.

Algoritmo 7.1 Algoritmo para la determinación exacta de la existencia de un camino alternativo

Requiere: Una planificación, y un arco (v, w) de dicha planificación

Produce: $NO\_FACTIBLE$ o $FACTIBLE$ según exista o no un camino alternativo entre v y w

si v es la última tarea de su trabajo, o w es la primera tarea de su trabajo **entonces**
  devuelve $FACTIBLE$;
fin si
$numero\_nodos\_visitados = 0$;
$nodos\_a\_visitar = \{SJ_v\}$;
$nodos\_considerados = \{SJ_v\}$;
mientras $nodos\_a\_visitar \neq \emptyset$ **hacer**
  si $numero\_nodos\_visitados \geq maxNodos$ **entonces**
    devuelve $NO\_FACTIBLE$;
  fin si
  si $PJ_w \in nodos\_a\_visitar$ **entonces**
    devuelve $NO\_FACTIBLE$;
  fin si
  $nodo\_actual =$ elegir un elemento de $nodos\_a\_visitar$;
  $nodos\_a\_visitar = nodos\_a\_visitar - \{nodo\_actual\}$;
  $numero\_nodos\_visitados = numero\_nodos\_visitados + 1$;
  si $r_{nodo\_actual} + p_{nodo\_actual} \leq r_{PJ_w}$ **entonces**
    si $\exists SM_{nodo\_actual}$, y $SM_{nodo\_actual} \notin nodos\_considerados$ **entonces**
      $nodos\_a\_visitar = nodos\_a\_visitar \cup \{SM_{nodo\_actual}\}$;
      $nodos\_considerados = nodos\_considerados \cup \{SM_{nodo\_actual}\}$;
    fin si
    si $\exists SJ_{nodo\_actual}$, y $SJ_{nodo\_actual} \notin nodos\_considerados$ **entonces**
      $nodos\_a\_visitar = nodos\_a\_visitar \cup \{SJ_{nodo\_actual}\}$;
      $nodos\_considerados = nodos\_considerados \cup \{SJ_{nodo\_actual}\}$;
    fin si
  fin si
fin mientras
devuelve $FACTIBLE$;

7.4.3. Condiciones de no mejora

Calcular o estimar el makespan de todos los vecinos es un proceso costoso computacionalmente, y por lo tanto es frecuente utilizar algunas condiciones de no mejora que se puedan evaluar rápidamente y que permitan descartar algunos vecinos y reducir el tiempo en la medida de lo posible.

El objetivo de esta sección es comprobar bajo qué condiciones puede llevar a una mejora de la función objetivo un movimiento de una tarea del interior de un bloque crítico hacia otra posición del interior de su mismo bloque crítico. La proposición 7.4.4 establece una condición suficiente para la no mejora de este tipo de movimientos, extendiendo la proposición 7.3.9 dada en la sección 7.3.5 para poder tratar con varias inversiones de órdenes de procesamiento en el mismo movimiento.

Proposición 7.4.4. *Sea H una planificación y $B = (b'\ v\ b\ w\ b'')$ un bloque crítico, en donde b, b' y b'' son secuencias de operaciones de la forma $b = (u_1 \dots u_n)$, $b' = (u'_1 \dots u'_{n'})$ y $b'' = (u''_1 \dots u''_{n''})$, y además $n' \geq 1$ y $n'' \geq 1$. Incluso aunque la planificación H' obtenida a partir de H moviendo w justo antes de v sea factible, H' no mejorará a H si la siguiente condición se cumple*

$$S_{u'_{n'}v} + S_{u_n w} + S_{w u''_1} \leq S_{u'_{n'}w} + S_{wv} + S_{u_n u''_1}. \tag{7.7}$$

Si $n = 0$, u_n debe ser sustituido por v en (7.7).

Demostración. Para todos los nodos v en H, como $r_v + p_v + q_v$ es la longitud del camino más largo desde el nodo *start* hasta el nodo *end* a través del nodo v, si el nodo v pertenece a un camino crítico, este valor es el makespan de H, o bien una cota inferior.

Sea C_{max} el makespan de la planificación H, C'_{max} el makespan de la planificación H', r_x y q_x la cabeza y la cola de la operación x en la planificación H, y r'_x y q'_x la cabeza y la cola de la operación x en la planificación H'. Entonces:

$$C_{max} = r_{u''_1} + p_{u''_1} + q_{u''_1}, \quad C'_{max} \geq r'_{u''_1} + p_{u''_1} + q'_{u''_1}$$

En H', como los arcos después de u''_1 no han cambiado, $q'_{u''_1} = q_{u''_1}$, y por lo tanto: $C'_{max} \geq C_{max}$ si $r'_{u''_1} \geq r_{u''_1}$ Ahora, hay que destacar que en H

$$r_{u''_1} = r_{u'_{n'}} + p_{u'_{n'}} + p_v + \sum_{x \in b} p_x + p_w + S_{u'_{n'}v} + S_{v u_1} + \sum_{i=1}^{n-1} S_{u_i u_{i+1}} + S_{u_n w} + S_{w u''_1}$$

Mientras que en H'

$$r'_{u''_1} \geq r'_{u'_{n'}} + p_{u'_{n'}} + p_w + p_v + \sum_{x \in b} p_x + S_{u'_{n'}w} + S_{wv} + S_{vu_1} + \sum_{i=1}^{n-1} S_{u_i u_{i+1}} + S_{u_n u''_1}$$

Y, en H', como los arcos anteriores a $u'_{n'}$ no han cambiado, $r'_{u'_{n'}} = r_{u'_{n'}}$, y por lo tanto, si la ecuación (7.7) se cumple, entonces $r'_{u''_1} \geq r_{u''_1}$ y entonces $C'_{max} \geq C_{max}$.

□

Se puede establecer un resultado análogo para los vecinos resultantes de mover la tarea v justo después de w.

Nótese que la demostración de la proposición 7.4.4 en el caso de la minimización del maximum lateness sería idéntica. Pero como ya ocurrió en la sección 7.3.5 con las proposiciones sobre no mejora presentadas allí, la generalización a las funciones objetivo de tipo suma es compleja, por lo que no vamos a proponer condiciones equivalentes para dichas funciones.

En esta sección hemos visto que en el SDST-JSP las inversiones de arcos críticos pertenecientes al interior de bloques críticos pueden producir mejora bajo determinadas condiciones. Por este motivo, para crear N^S decidimos ampliar la definición de N_2 para que incluya todos esos movimientos, siempre y cuando cumplan las citadas condiciones.

7.4.4. Definición de N^S

A partir de los resultados demostrados en las anteriores secciones proponemos una nueva estructura de vecindad, denominada N^S, y que se puede definir como sigue:

Definición 7.4 (N^S)**.** *Dada una planificación H, la vecindad $N^S(H)$ consiste en todas las planificaciones obtenidas a partir de H moviendo una operación v perteneciente a un bloque crítico B a otra posición de su mismo bloque crítico B, siempre y cuando la condición de no mejora* 7.4.4 *no se cumpla, y que se asegure la factibilidad del vecino aplicando los corolarios* 7.4.1 *y* 7.4.2*.*

Esta estructura de vecindad se podrá utilizar en la minimización del makespan y del maximum lateness. Para la minimización del weighted tardiness y del total flow time la vecindad se define de la misma forma, pero sin tener en cuenta la condición de no mejora de la proposición 7.4.4, debido a que no hemos demostrado resultados equivalentes para esas funciones objetivo.

Por lo tanto, N^S considera todos los bloques críticos de los caminos críticos, y para cada uno de los bloques B se trata de alterar el orden de las operaciones dentro de dicho

bloque. Cada operación se mueve a todas las otras posibles posiciones de su mismo bloque B (incluídos el principio y el final del bloque) y cada uno de estos movimientos se considera un vecino. Para establecer un esquema de vecindad eficiente, se utilizará en combinación con la comprobación de factibilidad detallada en la sección 7.4.2 y con la condición de no mejora explicada en la sección 7.4.3 (esta última condición sólo en el caso del makespan y del maximum lateness, ya que no la hemos definimos para las otras funciones objetivo).

7.5. Número de caminos críticos considerados

A lo largo de este capítulo hemos visto que las estrategias de vecindad propuestas consisten en invertir arcos críticos o cambiar la posición de alguna tarea en un bloque crítico. Sin embargo puede ocurrir que el grafo solución que representa una planificación tenga varios caminos críticos. Este hecho, aunque no será frecuente en las funciones objetivo de tipo bottleneck, ocurrirá prácticamente siempre en las funciones objetivo de tipo suma debido a la existencia de varios nodos finales. En esta sección realizaremos un breve estudio sobre cuántos caminos críticos considerar dependiendo de la función objetivo que queramos minimizar.

7.5.1. Caminos críticos en funciones objetivo de tipo bottleneck

Para empezar, hemos realizado una serie de experimentos para evaluar cuántos caminos críticos suele tener un grafo solución en la minimización del makespan. Experimentamos con 12 instancias de diferentes benchmarks, ejecutando una combinación de algoritmo genético y búsqueda tabú que ha generado aproximadamente diez millones de planificaciones para cada instancia. El resultado promedio es que el 89.38 % de las planificaciones tuvieron únicamente un camino crítico, el 9.12 % dos caminos críticos, el 0.95 % tres caminos críticos, y el 0.55 % cuatro o más caminos críticos. También hemos realizado experimentos minimizando el maximum lateness y los resultados son similares. Además, debemos tener en cuenta que incluso aunque existan varios caminos críticos, éstos pueden compartir la mayoría de las tareas.

En esta sección nos preguntamos si sería interesante utilizar el conjunto completo de vecinos de todos los caminos críticos cuando el grafo solución que representa una planificación H posea más de un camino crítico. El siguiente resultado, demostrado por Nowicki y Smutnicki en [105], justifica que en el JSP clásico es mejor considerar un único camino crítico

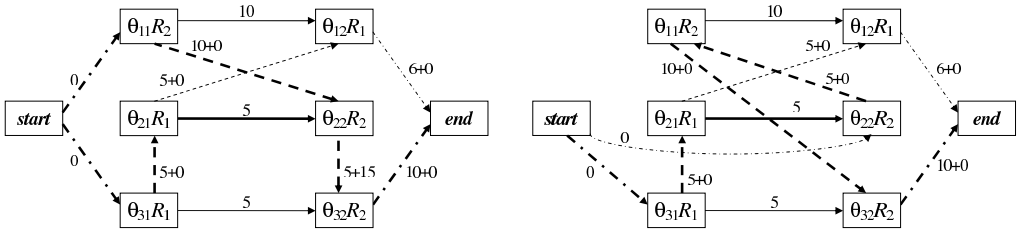
porque la mejora que se puede esperar de cualquier camino crítico es la misma. Sin embargo, como venimos observando, los resultados que se cumplen en el JSP clásico no acostumbran a ser ciertos en presencia de setups.

Proposición 7.5.1. *Sea H una planificación para una instancia del JSP, C_1 y C_2 dos caminos críticos en H, $N_1(C_1)$ y $N_1(C_2)$ los vecinos que devuelve N_1 de los caminos críticos C_1 y C_2, respectivamente, y $H' \in N_1(C_1) - N_1(C_2)$. Entonces, $C_{max}(H') \geq C_{max}(H)$.*

Este resultado sugiere que es mejor considerar únicamente los vecinos que se obtienen al invertir arcos que estén en todos los caminos críticos simultáneamente, y después elegir el vecino con el mejor makespan. Sin embargo, es importante tener en cuenta que la demostración se aplica al JSP clásico sin tiempos de setup, y en concreto a la estructura de vecindad N_1.

A continuación veremos que en el caso del SDST-JSP, y con las estructuras de vecindad propuestas en esta tesis, se pueden conseguir vecinos que mejoran sin necesidad de que todos los arcos críticos implicados pertenezcan a la intersección de todos los caminos críticos.

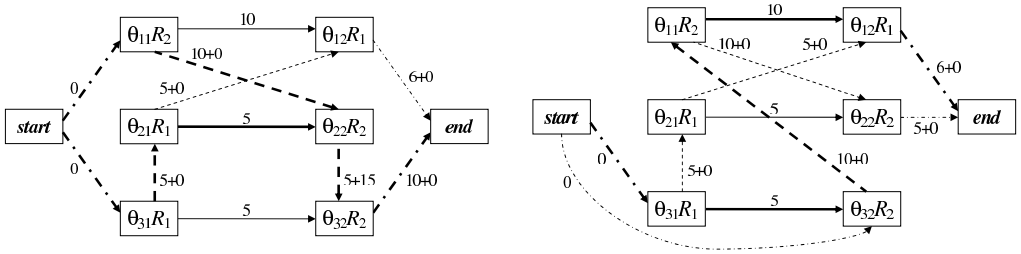
En la figura 7.6 presentamos un contraejemplo para la estructura de vecindad N_1^S. Se trata de una instancia con 3 trabajos y 2 recursos. La planificación de la figura 7.6 (a) tiene dos caminos críticos marcados en negrita, uno de ellos formado por las tareas $(\theta_{31}, \theta_{21}, \theta_{22}, \theta_{32})$, y el otro formado por las tareas $(\theta_{11}, \theta_{22}, \theta_{32})$, y su makespan es de 40. Consideramos el vecino resultante de invertir el arco crítico $(\theta_{11}, \theta_{22})$. Observemos que dicho vecino pertenece al segundo camino crítico, pero no al primero, aunque ambos caminos comparten la tarea θ_{22}. Mostramos en la figura 7.6 (b) el grafo solución de dicho vecino, y observamos que su makespan es 35. Sin embargo la desigualdad triangular de tiempos de setup no se cumple en esta instancia, ya que $S_{\theta_{22}\theta_{32}} = 15$, $S_{\theta_{22}\theta_{11}} = 0$ y $S_{\theta_{11}\theta_{32}} = 0$.



(a) Planificación inicial (b) Planificación vecina

Figura 7.6: Utilización de todos los caminos críticos con la estructura N_1^S

En la figura 7.7 presentamos otro contraejemplo, esta vez para la estructura de vecindad N^S. Se trata, de nuevo, de una instancia con 3 trabajos y 2 recursos similar a la anterior, pero esta vez la instancia cumple la desigualdad triangular de tiempos de setup. La planificación se muestra en la figura 7.7 (a) y tiene los mismos dos caminos críticos, uno de ellos formado por las tareas $(\theta_{31}, \theta_{21}, \theta_{22}, \theta_{32})$, y el otro formado por las tareas $(\theta_{11}, \theta_{22}, \theta_{32})$, y un makespan de 40. Consideramos el vecino resultante de insertar la tarea θ_{32} justo antes de la tarea θ_{11}. De nuevo, ese vecino pertenece al segundo camino crítico, pero no al primero, aunque ambos caminos críticos comparten dos de las tareas implicadas: θ_{22} y θ_{32}. La figura 7.7 (b) muestra el grafo solución del vecino, con un makespan de 31. En este caso, además, no es necesario que la instancia incumpla la desigualdad triangular de tiempos de setup, por lo que este hecho podría ocurrir en cualquier instancia del SDST-JSP.



(a) Planificación inicial

(b) Planificación vecina

Figura 7.7: Utilización de todos los caminos críticos con la estructura N^S

A través de los dos contraejemplos mostrados, hemos comprobado que en el SDST-JSP vecinos que pertenecen a un camino crítico pero no a otro camino crítico pueden producir una mejora de la función objetivo.

Por otra parte, una estrategia de escalada no es siempre la mejor elección al diseñar un criterio de aceptación, y muchas veces utilizaremos búsqueda tabú en nuestros experimentos, por lo que puede ser útil disponer de algunos vecinos que puedan igualar o incluso empeorar la solución actual. Además, en el caso de que la planificación tenga dos caminos críticos completamente disjuntos, puede ser interesante considerarlos todos, porque incluso aunque la planificación no vaya a mejorar inmediatamente, la estimación de la función objetivo probablemente nos permitirá continuar la búsqueda a través del vecino más prometedor.

Como consecuencia del análisis presentado en esta sección, hemos tomado la decisión de utilizar, en principio, todos los vecinos de todos los caminos críticos en las dos estructuras

de vecindad definidas. Sin embargo en el estudio experimental realizado en la sección 8.3.13 veremos que realmente las dos opciones ofrecen resultados muy similares, tanto en makespan como en tiempo de ejecución. Hemos realizado experimentos similares en la minimización del maximum lateness y los resultados también son similares. Esto era esperable, porque ya hemos visto que en este tipo de funciones objetivo el hecho de tener varios caminos críticos ocurre rara vez, por lo que considerarlos todos no puede producir una gran mejora.

7.5.2. Caminos críticos en funciones objetivo de tipo suma

En las funciones objetivo de tipo suma en principio se podrían considerar todos los caminos críticos posibles para generar vecinos, es decir, considerar todos los nodos end_i y además para cada uno de ellos considerar todos los caminos críticos que acaben en dicho nodo. Es claro que en este tipo de funciones objetivo un vecino de un camino crítico que acaba en un determinado nodo end_i puede producir una mejora sin necesidad de que comparta tareas con caminos críticos que acaben en nodos distintos.

Sin embargo, el aumento del número de caminos críticos, por el hecho de que cada trabajo define un extremo final de un camino crítico, tiene como consecuencia evidente que el punto crítico de estas funciones objetivo es el número de vecinos que se pueden considerar y que, si es excesivo, puede lastrar los algoritmos de búsqueda local. En el caso del weighted tardiness debemos recordar que sólo consideramos como caminos críticos los que acaben en los nodos de los trabajos i cuyo tiempo de finalización sea mayor que su due date d_i. Pero de todas formas si los due dates no son muy permisivos (o en el caso del total flow time, en el que no existen due dates) y el tamaño de la instancia es grande el número de caminos críticos puede ser a su vez muy elevado. El problema empeora si tenemos en cuenta que el algoritmo de estimación de vecinos será más costoso computacionalmente, como veremos en la sección 7.6.2.

Por estos motivos, propondremos estrategias de control para el tamaño de la vecindad, como por ejemplo considerar un número menor de caminos críticos. Una posibilidad es utilizar únicamente el camino crítico o caminos críticos del trabajo más importante, que en el caso del total flow time será simplemente el trabajo con el mayor tiempo de finalización, y en el caso del weighted tardiness el trabajo que más aporte a la función objetivo final. Una estrategia alternativa es considerar un número intermedio de caminos críticos, tomando por ejemplo la mitad de los trabajos de la instancia, eligiendo los más importantes según el criterio definido anteriormente. Para controlar el tamaño de la vecindad también podría

ser aconsejable utilizar la estructura N_1^S en lugar de la estructura N^S, ya que el número de vecinos que considera la primera estructura es mucho menor.

Las alternativas descritas reducen el número total de vecinos, y por lo tanto el tiempo total de ejecución. De todas formas con cualquiera de ellas es necesario utilizar algún mecanismo para evitar la generación de vecinos repetidos, ya que los caminos críticos hacia diferentes nodos end_i muchas veces tienen partes en común.

En la sección 10.3.2 presentamos un estudio experimental sobre la utilización de uno o todos los caminos críticos en la minimización del weighted tardiness, mientras que en la sección 11.2 presentamos un estudio similar para el caso del total flow time.

7.6. Estimación de la calidad de los vecinos

Calcular con exactitud la función objetivo de todos los vecinos para decidirse por uno de ellos es un proceso costoso computacionalmente. Una estrategia habitual para evitar el cálculo del valor exacto de la función objetivo de cada vecino consiste en el uso de estimadores. Proponemos aquí dos mecanismos de estimación: uno de ellos para las funciones makespan y maximum lateness, y el otro para las funciones weighted tardiness y total flow time. Además, demostraremos que los dos métodos siempre producen una cota inferior de la función objetivo en el caso de utilizar la estructura N_1^S, pero sin embargo éste no será el caso con la estructura N^S. También veremos que, en el caso de las funciones objetivo de tipo suma, la estimación será más costosa computacionalmente, además de menos precisa. Por último, comentaremos otros métodos de estimación existentes en la literatura.

7.6.1. Estimación en funciones objetivo de tipo bottleneck

El primer método para estimar el makespan tras un movimiento basado en la inversión de un arco crítico fue el propuesto por Taillard en [140]. Este método proporciona, en tiempo constante, una cota inferior del makespan de la nueva solución obtenida mediante el cálculo del camino más largo en el nuevo grafo solución que pasa por alguno de los nodos del arco invertido. Dell' Amico y Trubian en [46] generalizaron este método y propusieron un procedimiento denominado *lpath* para los vecinos generados con estructuras más complejas, en particular con la estructura NB, en la que se inspira N^S. A continuación proponemos un procedimiento *lpathS* que extiende el procedimiento *lpath* de forma que considere los tiempos de setup. El procedimiento se define inicialmente para la estimación de makespan,

aunque su adaptación al maximum lateness es inmediata.

lpathS, al igual que el procedimiento *lpath*, toma como entrada una secuencia de varias operaciones que requieren la misma máquina $(Q_1 \dots Q_q)$ tras un movimiento, donde $(Q_1 \dots Q_q)$ es una permutación de la secuencia de operaciones $(O_1 \dots O_q)$ que pertenecen a un mismo bloque crítico antes del movimiento. Sea r'_k y q'_k la cabeza y la cola de la operación k en la nueva planificación H'. Se sabe que $r'_k = r_k$ para todos los predecesores de O_1 en la planificación inicial H, y que $q'_k = q_k$ para todos los sucesores de O_q en H. Para cada $i = 1 \dots q$, *lpathS* estima el coste del camino más largo desde el nodo *start* hasta el nodo *end* que pasa a través de Q_i. El máximo de estos valores se toma como la estimación final del makespan de la planificación vecina.

La figura 7.2 muestra el pseudocódigo de este algoritmo, que es similar al método *lpath* pero añadiendo tiempos de setup. En el algoritmo denotamos por PM'_k y SM'_k al predecesor y sucesor en la máquina de la operación k en el nuevo grafo solución.

Algoritmo 7.2 Algoritmo de estimación *lpathS*

Requiere: Una secuencia de operaciones $(Q_1 \dots Q_q)$ tal como aparecen después de un movimiento

Produce: Una estimación del makespan de la planificación resultante

$a = Q_1$;
$r'_a = \max\{r_{PJ_a} + p_{PJ_a}, r_{PM'_a} + p_{PM'_a} + S_{PM'_a a}\}$;
para $i = 2$ hasta q **hacer**
  $b = Q_i$;
  $r'_b = \max\{r_{PJ_b} + p_{PJ_b}, r'_a + p_a + S_{ab}\}$;
  $a = b$;
fin para
$b = Q_q$;
$q'_b = \max\{q_{SJ_b} + p_{SJ_b}, q_{SM'_b} + p_{SM'_b} + S_{bSM'_b}\}$;
para $i = q - 1$ hasta 1 **hacer**
  $a = Q_i$;
  $q'_a = \max\{q_{SJ_a} + p_{SJ_a}, q'_b + p_b + S_{ab}\}$;
  $b = a$;
fin para
devuelve $\max_{i=1\dots q}\{r'_{Q_i} + p_{Q_i} + q'_{Q_i}\}$;

Este algoritmo se puede utilizar para estimar los vecinos creados con N_1^S, con N^S, y en general con cualquier vecindad basada en permutar operaciones de un único bloque crítico, sin más que elegir la secuencia de entrada $(Q_1 \dots Q_q)$ apropiada en cada caso. Para N_1^S, la secuencia es $(w\ v)$. Para N^S, si w se mueve antes que de v en un bloque de la forma $(b'\ v\ b\ w\ b'')$, entonces la secuencia de entrada es $(w\ v\ b)$, y si el vecino se crea moviendo v después de w la secuencia de entrada es $(b\ w\ v)$. El procedimiento *lpathS* es muy eficiente, y la gran mayoría de las veces la estimación coincide exactamente con el makespan real del vecino. La constatación experimental de este hecho se pondrá de manifiesto en el análisis experimental que presentamos en la sección 7.6.5. Para estimar el maximum lateness se puede utilizar el mismo procedimiento *lpathS* descrito en esta sección.

7.6.2. Estimación en funciones objetivo de tipo suma

Para estimar el weighted tardiness o el total flow time de un determinado vecino vamos a utilizar una extensión del algoritmo *lpathS* descrito en la sección 7.6.1. Al igual que con el makespan y el maximum lateness, este procedimiento toma como entrada una secuencia de varias operaciones de la forma $(Q_1 \dots Q_q)$ tras un movimiento, siendo $(Q_1 \dots Q_q)$ una permutación de la secuencia $(O_1 \dots O_q)$, que pertenecen a un mismo bloque crítico antes del movimiento.

Uno de los principales problemas que presenta la resolución de estas funciones objetivo es que el método de estimación utilizado antes con el makespan o el maximum lateness es ahora mucho más costoso computacionalmente. Esto es debido a que con las funciones objetivo de tipo bottleneck tenemos un único nodo final, y por lo tanto para cada una de las tareas afectadas debemos estimar una cabeza y una cola. Sin embargo en el caso del weighted tardiness tenemos tantos nodos finales como trabajos, por lo que ahora para cada una de las tareas afectadas debemos estimar una cabeza y sus N colas, siendo N el número de trabajos de la instancia. Por este motivo las estimaciones son más costosas, aunque se pueden seguir utilizando.

Denotamos las N colas de una tarea t por $q_t^1 \dots q_t^N$. Para cada $i = 1 \dots N$, *lpathSWT* estima el coste del camino más largo desde el nodo *start* hasta el nodo end_i a través de algún nodo incluído en $(Q_1 \dots Q_q)$. Después, se suman las estimaciones de todos los caminos para obtener la estimación final de la función objetivo de la solución vecina, siempre teniendo en cuenta los due dates y los pesos de cada trabajo en el caso del weighted tardiness. Al igual que en el método *lpathS*, para N_1^S, la secuencia de entrada es $(w\ v)$, y para N^S, si w se

mueve antes que de v en un bloque de la forma (b' v b w b''), la secuencia de entrada es (w v b), y si el vecino se crea moviendo v después de w la secuencia de entrada es (b w v).

El algoritmo 7.3 muestra el pseudocódigo del algoritmo utilizado. Respecto a *lpathS*, cambia la forma de estimar las colas de las tareas implicadas y el cálculo final de la estimación, que ahora depende del due date d_i y del peso w_i de cada trabajo i, y es una suma en lugar de un máximo. Recordemos que el total flow time se puede considerar como un caso particular del weighted tardiness, y por lo tanto podemos utilizar también el algoritmo 7.3, pero teniendo en cuenta que en este caso $d_i = 0\ \forall i$, y $w_i = 1\ \forall i$.

Algoritmo 7.3 Algoritmo de estimación *lpathSWT*

Requiere: Una secuencia de operaciones $(Q_1 \dots Q_q)$ tal como aparecen después de un movimiento

Produce: Una estimación del weighted tardiness de la planificación resultante

$EstTotal = 0$;
$a = Q_1$;
$r'_a = \max\{r_{PJ_a} + p_{PJ_a}, r_{PM'_a} + p_{PM'_a} + S_{PM'_a a}\}$;
para $i = 2$ hasta q **hacer**
  $b = Q_i$;
  $r'_b = \max\{r_{PJ_b} + p_{PJ_b}, r'_a + p_a + S_{ab}\}$;
  $a = b$;
fin para
para $i = 1$ hasta N **hacer**
  $b = Q_q$;
  $q'^i_b = \max\{q^i_{SJ_b} + p_{SJ_b}, q^i_{SM'_b} + p_{SM'_b} + S_{bSM'_b}\}$;
  para $j = q - 1$ hasta 1 **hacer**
    $a = Q_j$;
    $q'^i_a = \max\{q^i_{SJ_a} + p_{SJ_a}, q'^i_b + p_b + S_{ab}\}$;
    $b = a$;
  fin para
  $EstParcial = \max_{j=1\dots q}\{r'_{Q_j} + p_{Q_j} + q'^i_{Q_j}\}$;
  $EstTotal = EstTotal + (\max((EstParcial - d_i), 0) * w_i$;
fin para
devuelve $EstTotal$;

7.6.3. lpathS y lpathSWT producen una cota inferior al utilizar N_1^S

En esta sección presentaremos y demostraremos un resultado que revela que, utilizando la estructura de vecindad N_1^S, es decir invirtiendo un único arco crítico, los procedimientos *lpathS* y *lpathSWT* siempre devolverán una cota inferior de la función objetivo. La siguiente proposición se refiere a la minimización del makespan utilizando *lpathS*, pero se pueden demostrar resultados equivalentes para los demás casos, y la demostración sería idéntica.

Proposición 7.6.1. *La estimación que produce el procedimiento lpathS, aplicado a un vecino generado con la estructura de vecindad N_1^S, es una cota inferior del makespan de la nueva planificación.*

Demostración. Para demostrar que el procedimiento de estimación ofrece una cota inferior para el vecino resultante de invertir un único arco crítico (v, w), vamos a probar que dicho procedimiento calcula de forma exacta el camino más largo en el nuevo grafo que pasa por v o por w. Es importante destacar que en la nueva planificación H', únicamente se van a ver modificadas las cabezas de v y de los sucesores directos e indirectos que tenía en la planificación inicial H. En particular, $r'_z = r_z$, para todo z predecesor, directo o indirecto, de v en el grafo de la planificación inicial H. Por otra parte, sólo se van a ver modificadas las colas de w y de los predecesores directos e indirectos que tenía en la planificación inicial. En particular, $q'_z = q_z$, para todo z sucesor, directo o indirecto, de w en H. Observemos que una tarea x no puede ser simultáneamente predecesor y sucesor de otra tarea y, ya que eso implicaría un ciclo en el grafo solución. En la figura 7.8 mostramos el grafo que representa la planificación inicial y el grafo después de la inversión del arco crítico (v, w).

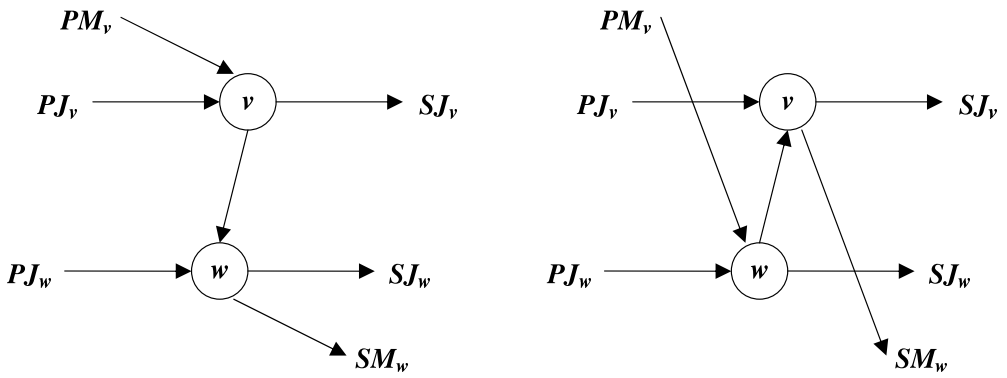


Figura 7.8: Inversión de un arco en un bloque crítico

El procedimiento *lpathS* primero estima la nueva cabeza de w de la siguiente forma:

$$r'_w = \text{máx}(r_{PJ_w} + p_{PJ_w}, r_{PM'_w} + p_{PM'_w} + S_{PM'_w w}) \tag{7.8}$$

Observamos que esta estimación de la nueva cabeza de w será exacta si r_{PJ_w} y $r_{PM'_w}$ no han cambiado de la planificación original a la planificación vecina. Sabemos que $r_{PM'_w}$ no ha cambiado debido a que $PM'_w = PM_v$, y por lo tanto en H es un predecesor tanto de v como de w. Por otra parte, r_{PJ_w} podría cambiar si PJ_w fuese sucesor directo o indirecto de v. Sin embargo, si esto ocurriera significaría que existe un camino desde v hasta PJ_w en el grafo original, y eso implicaría un ciclo en el grafo que representa la planificación vecina. Por lo tanto $r'_{PJ_w} = r_{PJ_w}$, y entonces r'_w es el valor exacto de la nueva cabeza de w.

A continuación se estima la nueva cabeza de v de la siguiente forma:

$$r'_v = \text{máx}(r_{PJ_v} + p_{PJ_v}, r'_w + p_w + S_{wv}) \tag{7.9}$$

Podemos observar que r_{PJ_v} no ha cambiado respecto a la planificación original, debido a que PJ_v, en la planificación original H, es un predecesor tanto de v como de w. Por otra parte, ya hemos comprobado que r'_w es exactamente el valor de la cabeza de w. Entonces, por estos dos motivos podemos concluir que r'_v también es el valor exacto de la cabeza de v.

Estos mismos razonamientos se pueden aplicar con la estimación de las nuevas colas. Empecemos con la nueva cola de v:

$$q'_v = \text{máx}(q_{SJ_v} + p_{SJ_v}, q_{SM'_v} + p_{SM'_v} + S_{vSM'_v}) \tag{7.10}$$

La estimación de la nueva cola de v será exacta si q_{SJ_v} y $q_{SM'_v}$ no han cambiado desde la planificación original a la planificación vecina. $q_{SM'_v}$ no ha cambiado debido a que $SM'_v = SM_w$, y por lo tanto en H es un sucesor tanto de v como de w. Por otra parte, q_{SJ_v} podría cambiar si SJ_v fuese predecesor directo o indirecto de w. Pero si esto ocurriera significaría que existe un camino desde SJ_v hasta w, y eso implicaría un ciclo en el grafo que representa la planificación vecina. Por lo tanto $q'_{SJ_v} = q_{SJ_v}$, y entonces q'_v es el valor exacto de la nueva cola de v.

En cuanto a la nueva cola de w:

$$q'_w = \text{máx}(q_{SJ_w} + p_{SJ_w}, q'_v + p_v + S_{wv}) \tag{7.11}$$

De nuevo, observamos que q_{SJ_w} no ha cambiado respecto a la planificación original, debido a que en la planificación original SJ_w es un sucesor tanto de v como de w. Además, ya hemos

comprobado que q'_v es exactamente el valor de la cola de v. Por estas dos razones, podemos concluir que q'_w también es el valor exacto de la cola de w.

Por último, el algoritmo de estimación calcularía el máximo entre la longitud del camino más largo que pasa a través de v y el que pasa a través de w, de esta forma:

$$Est. = \text{máx}(r'_v + p_v + q'_v, r'_w + p_w + q'_w) \tag{7.12}$$

Ya hemos comprobado que r'_v, q'_v, r'_w y q'_w son los valores exactos de las nuevas cabezas y colas de v y w, y por lo tanto este máximo nos daría la longitud exacta del camino más largo que pasa a través de v o w, lo que es claramente una cota inferior del makespan de la nueva solución. □

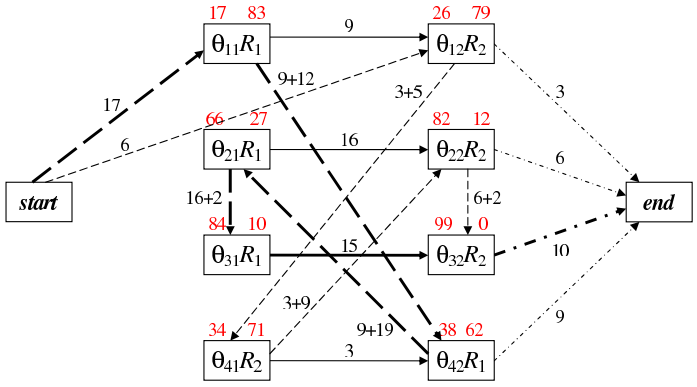
7.6.4. lpathS y lpathSWT no producen una cota inferior al utilizar N^S

Al contrario de lo que ocurre con N_1^S, al utilizar la estructura de vecindad N^S es posible que la estimación realizada sea mayor que la función objetivo real del individuo resultante. Esto puede ocurrir si hay un camino alternativo entre algunas de las operaciones del bloque crítico entre las que se realizará el movimiento. Observemos que esto no podía ocurrir con N_1^S ya que sólo podía existir un camino alternativo si el grafo contenía un ciclo. Para clarificar este hecho, en la demostración de la siguiente proposición mostraremos un ejemplo sobre la minimización del makespan en el SDST-JSP utilizando N^S y el procedimiento *lpathS*. Sin embargo, es posible demostrar este mismo resultado para las demás funciones objetivo, e incluso se pueden construir ejemplos con tiempos de setup nulos.

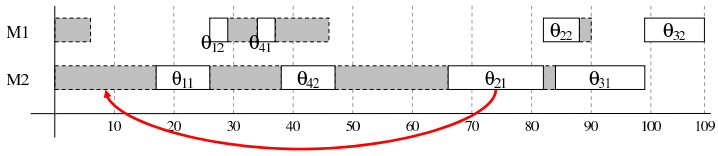
Proposición 7.6.2. *La estimación que produce el procedimiento lpathS, aplicado a un vecino generado con la estructura de vecindad N^S, puede ser mayor que el makespan de la nueva planificación.*

Demostración. Vamos a considerar la instancia del SDST-JSP y la planificación detalladas en la figura 7.9 (a). Es una instancia con 4 trabajos y 2 máquinas. Los dos números en rojo en la parte superior de cada una de las tareas indican la cabeza de la tarea y su cola, respectivamente. Además, las flechas en negrita indican el camino crítico de la planificación, con un makespan total de 109.

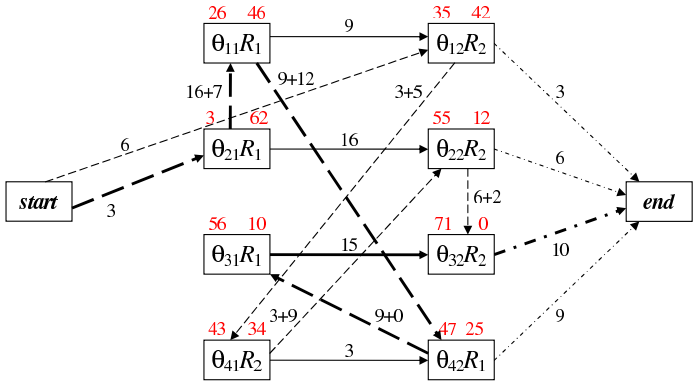
El camino crítico de la planificación es $\theta_{11}\theta_{42}\theta_{21}\theta_{31}\theta_{32}$, y las cuatro primeras tareas de ese camino crítico forman un bloque crítico de tamaño 4. El vecino que consideramos en este



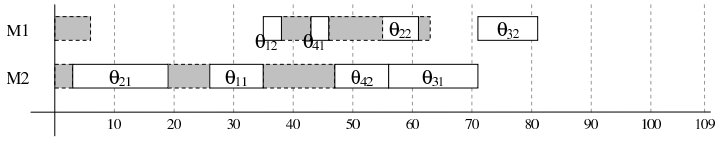
(a) Grafo disyuntivo de la planificación inicial



(b) Diagrama de Gantt de la planificación inicial



(c) Grafo disyuntivo de la planificación vecina



(d) Diagrama de Gantt de la planificación vecina

Figura 7.9: Ejemplo de una planificación vecina cuya estimación de makespan es 117 pero su makespan real es de 81

ejemplo es mover la tarea θ_{21} al principio de su bloque crítico. En la figura 7.9 (b) se muestra el diagrama de Gantt de la planificación, y la flecha roja indica el vecino considerado.

Ahora aplicamos el algoritmo de estimación $lpathS$:

$$r'_{\theta_{21}} = S_{0\theta_{21}} = 3$$

$$r'_{\theta_{11}} = r'_{\theta_{21}} + p_{\theta_{21}} + S_{\theta_{21}\theta_{11}} = 3 + 16 + 7 = 26$$

$$r'_{\theta_{42}} = \text{máx}(r'_{\theta_{11}} + p_{\theta_{11}} + S_{\theta_{11}\theta_{42}}, r_{\theta_{41}} + p_{\theta_{41}}) = \text{máx}(26 + 9 + 12, 34 + 3) = 47$$

$$q'_{\theta_{42}} = q_{\theta_{31}} + p_{\theta_{31}} + S_{\theta_{42}\theta_{31}} = 10 + 15 + 0 = 25$$

$$q'_{\theta_{11}} = \text{máx}(q'_{\theta_{42}} + p_{\theta_{42}} + S_{\theta_{11}\theta_{42}}, q_{\theta_{12}} + p_{\theta_{12}}) = \text{máx}(25 + 9 + 12, 79 + 3) = 82$$

$$q'_{\theta_{21}} = \text{máx}(q'_{\theta_{11}} + p_{\theta_{11}} + S_{\theta_{21}\theta_{11}}, q_{\theta_{22}} + p_{\theta_{22}}) = \text{máx}(82 + 9 + 7, 12 + 6) = 98$$

$$C'_{max} = \text{máx}(3 + 16 + 98, 26 + 9 + 82, 47 + 9 + 25) = 117$$

La estimación de makespan del vecino es 117, pero el individuo resultante tiene un makespan de 81, tal y como se muestra en el grafo disyuntivo y el diagrama de Gantt de las figuras 7.9 (c) y (d).

□

A través del ejemplo mostrado, podemos darnos cuenta de que el problema surge porque en el vecino original existía un camino alternativo, ajeno al camino crítico, entre las tareas θ_{11} y θ_{42}. Este hecho produjo que se estimara mal la cola de θ_{11}, ya que el algoritmo de estimación no tiene en cuenta que el camino desde el sucesor en el trabajo de θ_{11} (es decir θ_{12}) hasta el nodo *end* se ha reducido, debido a que en el vecino la tarea θ_{21} no está en dicho camino. Es decir, el algoritmo al calcular $q'_{\theta_{11}}$ no tiene en consideración que $q'_{\theta_{12}} \neq q_{\theta_{12}}$, y en consecuencia el algoritmo asigna a la cola de la tarea θ_{11} (y por tanto también a la de la tarea θ_{21} ya que el movimiento la coloca justo antes que θ_{11}) una sobre-estimación y eso lleva a que la estimación final del valor del makespan sea mayor que el makespan real del vecino.

7.6.5. Experimentos sobre la eficacia del algoritmo de estimación

La eficacia de los algoritmos de estimación para diferentes funciones objetivo propuestos en las secciones anteriores dependerá de muchos factores. En la tabla 7.1 se pueden ver los resultados de un estudio experimental realizado comparando las estimaciones de vecinos obtenidos con las estructuras de vecindad N_1^S y N^S (en las columnas), evaluando las diferentes funciones objetivo descritas (en las filas). Una de las instancias utilizadas es la ABZ7,

Tabla 7.1: Experimentos sobre la eficacia de los algoritmos de estimación propuestos

Función objetivo	Instancia	Estructura de vecindad N_1^S >	N_1^S =	N_1^S <	N^S >	N^S =	N^S <
Makespan	ABZ7	0.00 %	77.14 %	22.86 %	0.16 %	91.29 %	8.55 %
	ABZ7sdst	0.00 %	73.18 %	26.82 %	0.13 %	86.52 %	13.35 %
	i305_21	0.00 %	80.34 %	19.66 %	0.62 %	92.38 %	7.00 %
	t2-ps12	0.00 %	65.24 %	34.76 %	0.70 %	86.41 %	12.89 %
Maximum Lateness	ABZ7	0.00 %	76.72 %	23.28 %	0.14 %	90.89 %	8.97 %
	ABZ7sdst	0.00 %	73.22 %	26.78 %	0.14 %	86.68 %	13.18 %
	i305_21	0.00 %	79.79 %	20.21 %	0.60 %	92.25 %	7.15 %
	t2-ps12	0.00 %	65.72 %	34.28 %	0.85 %	87.20 %	11.95 %
Weighted Tardiness	ABZ7	0.00 %	46.54 %	50.46 %	0.28 %	70.71 %	29.01 %
	ABZ7sdst	0.00 %	51.83 %	48.17 %	0.24 %	66.45 %	33.31 %
	i305_21	0.00 %	44.15 %	55.85 %	1.21 %	62.02 %	36.77 %
	t2-ps12	0.00 %	26.07 %	73.93 %	1.44 %	43.16 %	55.40 %
Total Flow Time	ABZ7	0.00 %	48.46 %	51.54 %	0.31 %	68.05 %	31.64 %
	ABZ7sdst	0.00 %	52.03 %	47.97 %	0.33 %	64.63 %	35.04 %
	i305_21	0.00 %	39.62 %	60.38 %	1.40 %	51.92 %	46.68 %
	t2-ps12	0.00 %	26.38 %	73.62 %	1.40 %	41.58 %	57.02 %

tanto del JSP clásico como añadiendole tiempos de setup, para comprobar si el añadido de tiempos de setup aumenta la dificultad del proceso de estimación. También utilizaremos otras dos instancias con tiempos de setup: i305_21 y t2-ps12, para comprobar las diferencias entre instancias de diferentes benchmarks. Explicaremos con más detalle estas instancias y la forma en la que se generan los tiempos de setup para ellas en las secciones 8.2.1, 8.4.3 y 9.2.

Hemos realizado todos estos experimentos con un algoritmo genético hibridizado con búsqueda tabú, generando aproximadamente diez millones de vecinos en cada ejecución. En cada caso indicamos el porcentaje de vecinos generados cuya estimación es respectivamente mayor, igual, o menor, que la función objetivo real del vecino.

De los experimentos cuyos resultados se muestran en la tabla 7.1 se pueden extraer varias conclusiones, que enumeramos a continuación:

1. **La influencia de la estructura de vecindad:** la estructura de vecindad N^S ha obtenido en todos los casos un porcentaje bastante mayor de estimaciones correctas

que N_1^S. Promediando todos los casos, N_1^S ha obtenido un 57.90 % de estimaciones correctas, mientras que N^S ha obtenido un 73.88 %. Una posible explicación de este hecho es que en el caso de N_1^S el algoritmo estima el camino más largo del nuevo grafo que pasa a través de uno de los dos nodos implicados en el intercambio, mientras que en el caso de N^S el algoritmo generalmente estima más caminos que pasan a través de más nodos de la instancia, ya que suele haber más tareas implicadas en el movimiento.

2. **La influencia de la función objetivo:** podemos observar que, globalmente, en las funciones de tipo bottleneck se han estimado corréctamente entre el 65.24 % y el 92.38 % de los vecinos, mientras que en las funciones de tipo suma se han estimado correctamente entre el 26.07 % y el 70.71 % de los vecinos. En el caso particular del makespan se obtuvo una estimación exacta en el 81.56 % de los vecinos de media, en el maximum lateness también en el 81.56 %, en el weighted tardiness en el 51.37 % y en el total flow time en el 49.08 %. Observamos que en el makespan y el maximum lateness se obtienen resultados idénticos, y mucho mayores que los que se obtienen para las funciones de tipo suma. Por otra parte, comparando el weighted tardiness con respecto al total flow time no se puede concluir que una función sea más difícil de estimar que la otra, debido a que la diferencia entre los porcentajes es muy pequeña. La única conclusión en este aspecto es que el algoritmo de estimación para las funciones de tipo suma, además de ser más complejo computacionalmente, es capaz de obtener el valor exacto de la función objetivo un porcentaje bastante menor de las veces.

3. **La influencia de los tiempos de setup:** hemos considerado el problema ABZ7 tanto en presencia de tiempos de setup como sin ellos, y en seis de los ocho casos los porcentajes de estimación exacta son ligeramente inferiores en presencia de setups. Promediando todos los casos, en la instancia ABZ7 se ha conseguido una estimación igual a la función objetivo en el 71.23 % de los vecinos, mientras que en la instancia ABZ7sdst se ha conseguido en el 69.32 % de los vecinos. Sería necesario un estudio experimental con más instancias para poder realizar una conclusión fiable, ya que un 2 % de diferencia en una única instancia no es suficiente para poner de manifiesto que los tiempos de setup añaden una cierta dificultad al proceso de estimación.

4. **La influencia de la instancia del problema:** también podemos concluir que la función objetivo puede ser más o menos difícil de estimar correctamente dependiendo de la estructura de cada instancia en particular. En esta serie de experimentos la

instancia t2-ps12 ha obtenido porcentajes de estimación exacta menores que el resto de instancias, para todas las funciones objetivo consideradas. En la minimización del weighted tardiness, dicha instancia únicamente ha conseguido la estimación exacta en un 26.07 % de los vecinos. Por otra parte, la instancia i305_21 ha resultado ser la más sencilla de estimar correctamente en el caso del makespan y del maximum lateness, obteniendo porcentajes de estimación exacta entre el 79.79 % y el 92.38 %, mientras que han sido las instancias ABZ las que obtienen mayores porcentajes de estimación correcta en el caso del weighted tardiness y el total flow time, con valores entre el 46.54 % y el 70.71 %.

5. **Estimación mayor que el valor real:** Cuando utilizamos la estructura de vecindad N_1^S este hecho nunca ocurre, lo cual es coherente con la proposición demostrada en la sección 7.6.3. En cuanto a la estructura N^S, el porcentaje de vecinos para los que el algoritmo de estimación obtiene un valor mayor que el valor real de la función objetivo es siempre muy reducido. Entre todas las pruebas realizadas con esta estructura, ha ocurrido de media en el 0.62 % de los vecinos, y en el peor de los casos sólo supuso el 1.44 % de los vecinos.

Hemos comprobado que en el caso del total flow time y del weighted tardiness la proporción de casos en los que la estimación es menor que el valor real de la función objetivo es bastante alta. A partir de las conclusiones extraídas de estos experimentos, pensamos que una decisión adecuada para estas dos funciones puede ser evaluar los vecinos de forma exacta cuando su estimación sea menor que el valor de la función objetivo de la planificación original. Sin embargo, la utilización de estimaciones nos permitirá descartar todos los vecinos cuya estimación sea mayor o igual que dicho valor. A partir de los resultados experimentales mostrados en la sección 10.3.1, concluiremos que la mejora obtenida de esta forma compensa ampliamente el tiempo de ejecución adicional utilizado.

Por otra parte, en el caso del makespan y del maximum lateness, ya que las estimaciones son mucho más precisas, confiaremos plenamente en ellas y únicamente consideraremos el individuo con la mejor estimación. En la sección 8.3.9 comprobaremos de forma experimental que esta es la mejor opción para esas dos funciones.

7.6.6. Otros algoritmos de estimación

El problema de que los algoritmos de estimación descritos en esta sección no ofrezcan buenos resultados en las funciones objetivo de tipo suma ha sido puesto de manifiesto recientemente en [86]. En este trabajo Mati et al. tratan con diversas funciones objetivo, algunas de ellas de tipo suma, y su forma de resolver ese problema es proponer un algoritmo de estimación más preciso. Recordemos que los algoritmos de estimación descritos en esta tesis consideran los caminos que en el nuevo grafo pasan a través de alguno de los nodos de la porción de bloque crítico que hemos modificado en el movimiento. Mati et al. proponen considerar también en la estimación otros caminos del grafo, ajenos al bloque crítico modificado, para los que sea conocido de antemano que no van a cambiar en el nuevo vecino. En concreto, dado el vecino resultante de invertir un arco crítico (v, w), proponen considerar también en la estimación los caminos más largos del grafo origen que pasan a través de los nodos u que cumplen que $u \neq v$ y $l_u = l_v$, siendo l_u el nivel de u en el grafo de la planificación original, es decir, el máximo número de arcos en un camino desde el nodo inicial hasta el nodo u. Essafi et al. en [49] tratan de minimizar el weighted tardiness en el JSP y, basándose en los trabajos de Mati et al., también utilizan dicho algoritmo de estimación más elaborado.

Sin embargo consideramos que su propuesta tiene dos inconvenientes: el primero de ellos es que es un método de estimación más complejo y más costoso computacionalmente. Por ejemplo, en el caso de la inversión de un único arco crítico, la complejidad del algoritmo de estimación para estimar un camino desde el nodo inicial hasta uno de los nodos finales aumenta de $O(1)$ a $O(N)$, siendo N el número de trabajos (ver [86]). El otro inconveniente es que, aunque su función de estimación sea más precisa que las utilizadas en esta tesis, todavía esta lejos de ser muy precisa. Mati et al. realizan un estudio experimental sobre la precisión de su método, y en el caso concreto del weighted tardiness el porcentaje de veces que su estimación coincide con el valor real de la función objetivo se sitúa entre el 57 % y el 76 % dependiendo del tamaño de la instancia. Como hemos visto en la tabla 7.1 presentada en la sección 7.6.5, nosotros hemos obtenido el valor real de la función objetivo entre el 26 % y el 52 % de las veces cuando consideramos la inversión de un único arco crítico. Por lo tanto aunque la precisión aumente considerablemente, todavía esta bastante lejos del 100 %.

Por estos motivos pensamos que en las funciones objetivo de tipo suma puede ser una alternativa mucho más eficiente utilizar el método de estimación más rápido posible, para descartar un gran número de vecinos en un tiempo muy reducido, y posteriormente calcular de forma exacta la función objetivo de los restantes vecinos.

7.7. Aplicación de un movimiento de la vecindad

Una vez construída toda la vecindad de una solución, elegiremos uno de los vecinos basándonos en las estimaciones de todos los vecinos. Una vez elegido uno ya podemos construir el nuevo grafo solución, y posteriormente calcular su valor exacto de la función objetivo y determinar sus caminos críticos. Sin embargo, un cálculo de la función objetivo desde cero sería un proceso costoso computacionalmente. Sin embargo, como veremos en las siguientes secciones, esto no es necesario ya que la gran mayoría del nuevo grafo solución es exactamente igual al del individuo original, por lo que podemos omitir la mayoría de cálculos. Estas consideraciones también son explicadas con detalle por Mattfeld en [88].

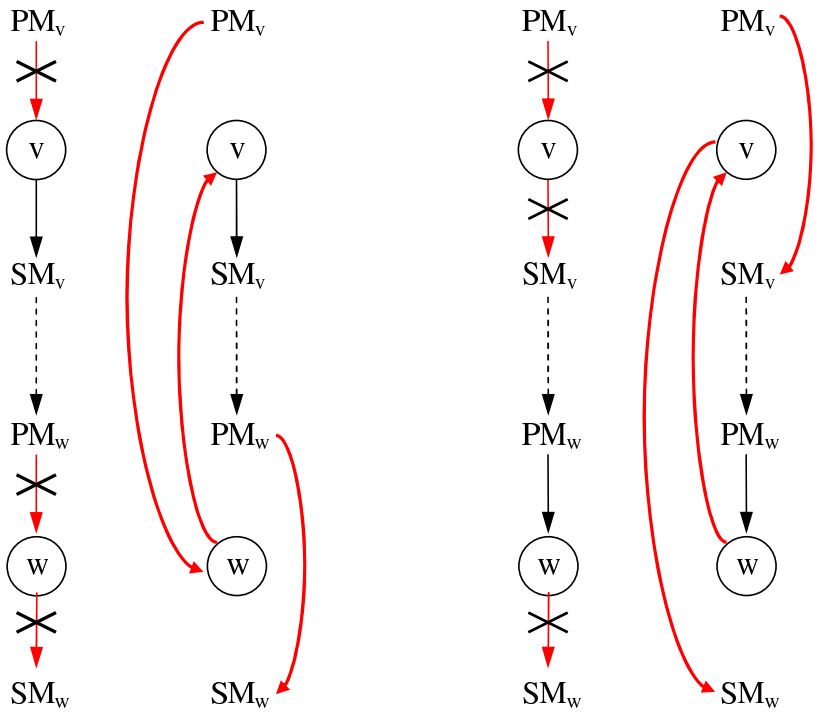
7.7.1. Construcción del nuevo grafo solución

El nuevo grafo solución será idéntico al del individuo original excepto algunos de los arcos del bloque crítico modificado. Si tenemos un bloque crítico de la forma (b' v b w b''), donde b, b' y b'' son secuencias de operaciones, pudiendo ser cualquiera de ellas vacía, y queremos construir un vecino moviendo la tarea w justo antes de la tarea v aplicaremos los siguientes cambios:

1. Borrar el arco (PM_w, w).
2. Si $\exists PM_v$ borrar el arco (PM_v, v) y añadir el arco (PM_v, w).
3. Si $\exists SM_w$ borrar el arco (w, SM_w) y añadir el arco (PM_w, SM_w).
4. Añadir el arco (w, v).

Los cambios indicados también se pueden ver de forma gráfica en la figura 7.10 (a). En el grafo de la izquierda se indican los arcos que se borran y en el de la derecha los que se añaden. Se puede utilizar un razonamiento similar para los vecinos creados moviendo la tarea v justo después de w. En ese caso se aplicarán los siguientes cambios:

1. Borrar el arco (v, SM_v).
2. Si $\exists SM_w$ borrar el arco (w, SM_w) y añadir el arco (v, SM_w).
3. Si $\exists PM_v$ borrar el arco (PM_v, v) y añadir el arco (PM_v, SM_v).
4. Añadir el arco (w, v).



(a) Moviendo w antes de v (b) Moviendo v después de w

Figura 7.10: Construcción del nuevo grafo solución una vez elegido uno de los vecinos

Mostramos estos cambios de forma gráfica en la figura 7.10 (b). Al igual que antes, el grafo de la izquierda indica los arcos que se borran y el de la derecha los que se añaden.

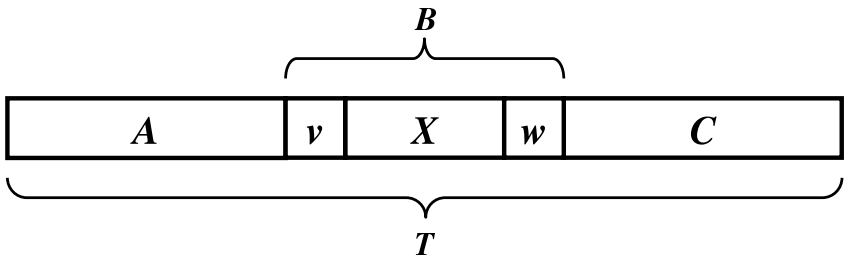
7.7.2. Cálculo del nuevo orden topológico

El orden topológico de la solución vecina será en general muy parecido al de la solución inicial, y determinarlo será necesario para el cálculo eficiente de la función objetivo de la nueva planificación. A continuación describiremos el método propuesto por Mattfeld en [88] para calcularlo. Si tenemos un vector T con el orden topológico del individuo original, para obtener el orden topológico T' del vecino sólo deberemos modificar la porción de T comprendida entre la primera tarea v y la última tarea w afectadas en el movimiento, siendo el resto del vector idéntico en T y en T'; denominaremos B a esta porción de T. Por otra parte, denominaremos A a la porción comprendida desde el principio de T hasta B, y C a la porción comprendida desde B hasta el final de T. La figura 7.11 (a) muestra el vector T.

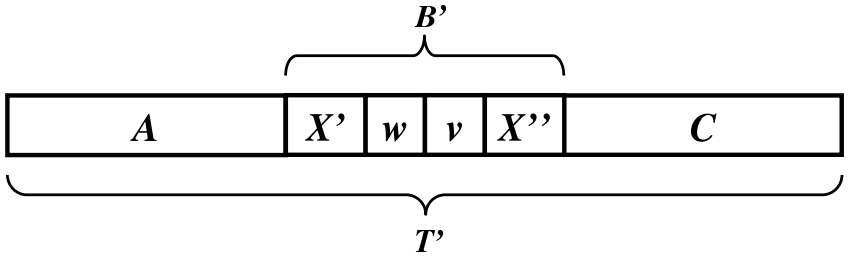
Si consideramos los nodos de $X = B - \{v, w\}$, su posición en el vector se debe a una de las siguientes razones:

1. Un nodo de X aparece antes que w en T porque es un predecesor directo o indirecto de w.
2. Un nodo de X aparece después que v porque es un sucesor directo o indirecto de v.
3. Un nodo de X no esta relacionado ni con v ni con w.

En el nuevo orden topológico T', las porciones A y C permanecen iguales, pero debemos permutar las operaciones de B para construir B'. El nodo w ahora será un predecesor directo del nodo v, por lo que en T' colocamos w antes que v. Para el resto de nodos distinguiremos entre el caso 1 y el caso 2, dividiendo X en X' y X'' respectivamente. Los nodos del caso 3 pueden aparecer en X' o en X'' indistintamente, ya que esos nodos de X no son sucesores de v ni predecesores de w. Para construir X' y X'' etiquetamos los nodos de X que son predecesores directos o indirectos de w en la nueva planificación, e incluímos dichos nodos en X'. Los nodos no etiquetados formarán X'' y serán o bien sucesores de v o bien no relacionados con v ni con w. La figura 7.11 (b) muestra el vector T', que será el orden topológico de la planificación vecina.



(a) Vector de la planificación original



(b) Vector de la planificación vecina

Figura 7.11: Vectores con la ordenación topológica de una planificación

Según algunos resultados experimentales mostrados por Mattfeld en [88] sobre varias instancias de benchmarks típicos del JSP, la porción de T que se debe modificar para conseguir T' es aproximadamente del 5 % del tamaño total de T. Por lo tanto el ahorro de tiempo al calcular de esta forma la nueva ordenación topológica es considerable.

7.7.3. Cálculo de las nuevas cabezas y colas

A partir del nuevo orden topológico T' ya podemos calcular de forma más eficiente las nuevas cabezas y colas de las operaciones en el nuevo grafo solución. En particular, sólo necesitamos recalcular las cabezas r_v, $v \in \{B' \cup C\}$, ya que las cabezas de los nodos de A no cambian. De la misma forma, sólo recalcularemos las colas q_v, $v \in \{B' \cup A\}$, ya que las colas de los nodos de C no cambian. Esta técnica permite ahorrar una cantidad importante de tiempo de ejecución en el cálculo de la nueva función objetivo, cantidad cercana al 50 % según [88].

7.8. Conclusiones

Hemos comenzado este capítulo demostrando varias propiedades que se cumplen en el JSP pero que dejan de cumplirse en el SDST-JSP. Un resultado muy importante sobre las estructuras de vecindad es que ninguna estructura basada en intercambios de tareas pertenecientes a bloques críticos puede cumplir la propiedad de conectividad. Esto es un grave problema para los algoritmos de resolución exacta, pero en el caso de un método heurístico o aproximado dicha propiedad no es imprescindible. También hemos demostrado que, al contrario de lo que ocurre en el JSP, la inversión de un único arco crítico puede dar lugar a una planificación no factible. Por otra parte, en el caso del SDST-JSP la inversión de un arco situado en el interior de un bloque crítico puede dar lugar a una planificación con un valor menor de su función objetivo. Estas consideraciones hacen más compleja la resolución del SDST-JSP que la del JSP sin tiempos de setup.

Posteriormente nos hemos planteado la extensión de dos estructuras de vecindad clásicas del JSP, una basada en la inversión de un único arco crítico, y la otra en la inserción de una tarea en otro lugar de su mismo bloque crítico. Hemos extendido primero la estructura del JSP denominada N_1 por Mattfeld en [88], denominándola N_1^S. Esta estructura considera inversiones de un único arco crítico en cada vecino. Para definir una estructura eficiente, hemos considerado todas las posibles inversiones de un único arco crítico y hemos detallado

condiciones de factibilidad y de no mejora para dichas inversiones. Posteriormente hemos propuesto otra estructura de vecindad denominada N^S que considera inversiones de varios órdenes de procesamiento de tareas críticas en el mismo movimiento. Para ello, hemos extendido las condiciones de factibilidad y no mejora desarrolladas para la estructura N_1^S. En general, se podrán utilizar las dos estructuras de vecindad diseñadas en este capítulo para minimizar cualquiera de las funciones objetivo estudiadas en esta tesis, siempre y cuando se tenga en cuenta el cambio en el modelo del grafo disyuntivo y lo que ello implica en las condiciones anteriores, especialmente en las de no mejora.

A continuación hemos realizado un estudio sobre el número de caminos críticos considerados para crear vecinos. Dividimos este estudio según que la función objetivo sea de tipo bottleneck o de tipo suma, ya que los resultados podrían ser distintos. Comprobamos mediante dos ejemplos que en el SDST-JSP, un vecino que pertenece a un camino crítico pero no a otro puede producir una mejora de la función objetivo. Proponemos también en este capítulo métodos para estimar la función objetivo de los vecinos creados, que son una extensión de métodos similares para el JSP clásico. Demostramos que la estimación que proporcionan estos métodos es una cota inferior en el caso de vecinos creados mediante la estructura N_1^S, pero esto no es así en el caso de vecinos creados mediante N^S. Además, hemos comprobado experimentalmente que el porcentaje de estimaciones exactas realizadas por este algoritmo depende de muchos factores, siendo el más influyente la función objetivo que se desea minimizar. Para las funciones objetivo de tipo bottleneck la estimación es muy rápida y precisa, pero para las funciones objetivo de tipo suma es menos precisa además de más costosa computacionalmente. Por último, describimos una forma eficiente de calcular la función objetivo real del vecino elegido finalmente, sin necesidad de recalcular las cabezas y las colas de todas las operaciones.

En general, el estudio realizado en este capítulo es una de las principales aportaciones de esta tesis.

Capítulo 8

ESTUDIO EXPERIMENTAL. MINIMIZACIÓN DEL MAKESPAN

8.1. Introducción

En este capítulo realizamos un estudio experimental sobre la minimización del makespan. En primer lugar definiremos los benchmarks utilizados. Posteriormente realizaremos un ajuste de parámetros para elegir la mejor configuración posible para nuestros métodos. Una vez decidida la configuración, compararemos nuestros resultados con los mejores métodos conocidos en la literatura para el SDST-JSP. Por último, completaremos el estudio experimental con resultados sobre benchmarks típicos del JSP clásico sin tiempos de setup.

8.2. Descripción de los benchmarks utilizados

A lo largo de este estudio experimental utilizaremos varios benchmarks del SDST-JSP: el conocido conjunto BT, el conjunto de instancias utilizadas por Cheung y Zhou en [40], y el conjunto de instancias propuesto por Vela et al. en [151].

8.2.1. El conjunto BT

Estas instancias, propuestas por Brucker y Thiele en [33], se derivan de las correspondientes instancias de Adams et al. propuestas en [3] para el JSP, pero introduciendo tiempos de setup. Cada instancia se caracteriza mediante su número de trabajos, su número de máquinas, y una matriz de tipos de setup para las operaciones. Cada operación tiene un tipo de setup y todas las operaciones del mismo trabajo tienen el mismo tipo de setup. La duración de las operaciones se ha generado aleatoriamente entre 1 y 100. La matriz de tiempos de setup indica el tiempo necesario para cambiar de un tipo de setup a otro en cualquier máquina. Por tanto, cada instancia se identifica mediante una terna (trabajos $\times$ máquinas $\times$ tipos_setup). Hay un total de 15 instancias llamadas desde t2-ps01 hasta t2-ps15, divididas en tres grupos. Las instancias t2-ps01 hasta t2-ps05 son del tipo $10 \times 5 \times 5$ (instancias pequeñas). Las instancias desde t2-ps06 hasta t2-ps10 son del tipo $15 \times 5 \times 5$ (instancias medianas). Y las instancias t2-ps11 hasta t2-ps15 son del tipo $20 \times 5 \times 10$ (instancias grandes).

Los tiempos de setup se han generado basándose en un cociente q de tiempo de setup medio dividido por el tiempo de procesamiento medio. Los dos primeros grupos de instancias tienen un cociente q cercano a 0.5, que corresponde a tiempos de setup medios, mientras que el tercer grupo tiene un cociente q cercano a 1, obteniendo tiempos de setup más elevados. Los tiempos de setup tienen una estructura especial, como veremos a continuación. La matriz de tiempos de setup para las instancias con 5 tipos de setup es la siguiente, para todas las instancias:

$$\begin{pmatrix} 10 & 20 & 30 & 40 & 50 \\ 0 & 10 & 20 & 30 & 40 \\ 40 & 0 & 10 & 20 & 30 \\ 30 & 40 & 0 & 10 & 20 \\ 20 & 30 & 40 & 0 & 10 \\ 10 & 20 & 30 & 40 & 0 \end{pmatrix}$$

La primera fila de la matriz corresponde a los tiempos de setup iniciales de la máquina, según el primer trabajo ejecutado en dicha máquina. El resto de filas indican el tiempo de setup necesario para pasar desde el trabajo indicado por el número de fila hasta el trabajo indicado por el número de columna. Si la instancia tiene más trabajos que columnas tiene la matriz, simplemente se vuelve a comenzar por la primera fila o columna según sea necesario. Por otra parte, la matriz de tiempos de setup para las instancias con 10 tipos de setup es la

siguiente, para todas las instancias:

$$\begin{pmatrix}
10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 & 100 \\
0 & 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 \\
90 & 0 & 10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 \\
80 & 90 & 0 & 10 & 20 & 30 & 40 & 50 & 60 & 70 \\
70 & 80 & 90 & 0 & 10 & 20 & 30 & 40 & 50 & 60 \\
60 & 70 & 80 & 90 & 0 & 10 & 20 & 30 & 40 & 50 \\
50 & 60 & 70 & 80 & 90 & 0 & 10 & 20 & 30 & 40 \\
40 & 50 & 60 & 70 & 80 & 90 & 0 & 10 & 20 & 30 \\
30 & 40 & 50 & 60 & 70 & 80 & 90 & 0 & 10 & 20 \\
20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 & 0 & 10 \\
10 & 20 & 30 & 40 & 50 & 60 & 70 & 80 & 90 & 0
\end{pmatrix}$$

Brucker y Thiele en [33] y Artigues y Feillet en [15], comentan que esas mismas instancias sin los tiempos de setup (LA01 hasta LA15) se resuelven fácilmente mediante sus métodos de ramificación y poda. También hemos resuelto estas instancias de forma óptima mediante el algoritmo genético propuesto por Gonzalez et al. en [61]. El mayor interés que tiene utilizar este benchmark es que es uno de los más utilizados en la literatura, y ya existen soluciones muy buenas o incluso óptimas para muchas de las instancias.

8.2.2. Benchmark propuesto por Cheung y Zhou

En este estudio experimental también presentaremos resultados sobre el conjunto de instancias propuesto por Cheung y Zhou en [40]. Éste es un conjunto de 45 instancias de tamaños 10×10, 20×10 y 20×20, y organizados en 3 tipos. Las instancias de tipo 1 tienen tiempos de procesamiento y tiempos de setup uniformemente distribuídos en (10, 50). Las instancias de tipo 2 tienen tiempos de procesamiento en (10, 50) y tiempos de setup en (50, 99). Las instancias de tipo 3 tienen tiempos de procesamiento en (50, 99) y tiempos de setup en (10, 50). Estas instancias se denominan desde ZRD01 hasta ZRD45, y hay que destacar que la desigualdad triangular de tiempos de setup no se cumple para estas instancias.

8.2.3. Benchmark propuesto por Vela et al.

Las instancias del conjunto BT tienen tamaños 10×5, 15×5 y 20×5. Para poder experimentar con instancias mayores que las del conjunto BT, utilizaremos también el benchmark

definido por Vela et al. en [151]. Su definición está motivada por el hecho de que muchas de las instancias BT son relativamente fáciles de resolver, y este benchmark tiene mucho interés por el hecho de que sus instancias se derivan de las del conjunto de 10 instancias elegidas por Applegate y Cook en [9] como difíciles de resolver para el JSP clásico. Su tamaño varía desde 15×10 para las más pequeñas (LA21, LA24 y LA25), 20×10 para las LA27 y LA29, 15×15 para las LA38 y LA40, y 20×15 para las más grandes (ABZ7, ABZ8 y ABZ9). Estas instancias se extienden al SDST-JSP utilizando el mismo criterio que las instancias BT, es decir, los tiempos de setup dependen únicamente de los trabajos y se obtienen de una de las dos matrices detalladas en la sección 8.2.1, que identifican 5 ó 10 tipos de tiempos de setup. Las instancias con 15 trabajos son de tipo 5 y las instancias con 20 trabajos son de tipo 10. Cada instancia se identifica con el mismo nombre de la instancia JSP seguido por *sdst*. Este benchmark se puede descargar del sitio web *http://www.aic.uniovi.es/tc/spanish/repository.htm*.

8.3. Configuración óptima para los métodos propuestos

8.3.1. Introducción

En esta sección nuestro objetivo es buscar la configuración más adecuada para los métodos propuestos, con el objetivo de comparar nuestros algoritmos con los mejores métodos conocidos en la literatura. Para ello consideraremos inicialmente un híbrido de algoritmo genético y búsqueda local de tipo máximo gradiente, y elegiremos un tiempo de ejecución suficiente para que dicho método converja adecuadamente. Una vez hecho esto, en las sucesivas secciones modificaremos diversos aspectos del algoritmo para intentar mejorar los resultados obtenidos, tratando de ajustarnos en la medida de lo posible al tiempo de ejecución elegido inicialmente. El primer paso será comparar las estructuras de vecindad propuestas en esta tesis para quedarnos con la que mejores resultados ofrezca. Posteriormente veremos la influencia del constructor de planificaciones utilizado. A continuación exploraremos si el algoritmo híbrido presenta el efecto de sinergia perseguido comparando sus resultados con los de los métodos por separado. En la siguiente serie de experimentos introduciremos la búsqueda tabú, y comprobaremos su eficacia tanto en su utilización de forma aislada como en la combinación con un algoritmo genético.

Despúes de los experimentos anteriores habremos concluído que la mejor opción es una combinación de algoritmo genético y búsqueda tabú. Entonces realizaremos varios estudios adicionales sobre aspectos mucho más concretos de dicho método, entre otros el hecho de

utilizar o no un operador de mutación en el genético, la forma de utilizar el algoritmo de estimación, la cantidad de caminos críticos que se deben considerar, o si se debe realizar o no una comprobación exacta de la factibilidad de los vecinos creados.

Para todos los estudios realizados en esta fase de calibración utilizaremos el siguiente conjunto de 15 instancias: las 5 instancias más grandes del conjunto BT (t2-ps11 a t2-ps15), y las 10 instancias del benchmark propuesto por Vela et al. en [151] y que ya describimos en la sección anterior. De esta forma tenemos un conjunto de prueba con un número aceptable de instancias y con tamaños bastante variados. No hemos incluído las 10 instancias más pequeñas del conjunto BT porque hemos comprobado que su resolución es bastante más sencilla, y por tanto en las comparaciones entre diferentes configuraciones no aportan mucha información adicional. Sin embargo, una vez que obtengamos una buena configuración de parámetros para nuestros algoritmos, mostraremos también resultados sobre esas instancias y sobre varios benchmarks más.

En los experimentos realizados a lo largo de este capítulo hemos realizado 30 ejecuciones para cada una de las instancias, y en las tablas de resultados indicaremos siempre el mejor resultado alcanzado, el número de veces que se alcanza dicho resultado, y el resultado medio en las 30 ejecuciones. Además, en los casos en los que sea más relevante, indicaremos la desviación estándar. Otro dato importante, y que por tanto siempre tendremos en cuenta, es el tiempo de ejecución utilizado.

Dada la extensión del análisis experimental realizado en este capítulo, no incluímos en él tests estadísticos que avalen las conclusiones de los experimentos y las decisiones que de ellas se derivan. En el capítulo 9 sí que incluímos un estudio estadístico pormenorizado, realizado en parte con el apoyo de la Unidad de Consultoría Estadística de la Universidad de Oviedo, debido a que una parte importante del análisis experimental no se repite, y en el que además se analizan los resultados sobre un banco de ejemplos de un tamaño mucho mayor, cuyas instancias se han generado en función de cuatro parámetros distintos.

8.3.2. Ajuste inicial de parámetros

Proponemos inicialmente una combinación de algoritmo genético y búsqueda local de tipo máximo gradiente, utilizando un constructor estándar de planificaciones semiactivas y la estructura de vecindad N^S. El objetivo de esta primera serie de experimentos es comprobar si dicho método converge adecuadamente. Para ello, hemos elegido arbitrariamente una población de tamaño 200, debido a que es un valor típico en la literatura. Decidimos ejecutar

este algoritmo durante un número elevado de generaciones (600) y comprobaremos los valores de makespan medio y tiempo consumido cada 100 generaciones, para decidir cuándo el algoritmo alcanza una convergencia adecuada.

En la tabla 8.1 se presentan los resultados de estos experimentos. En concreto se muestra el makespan medio alcanzado en las 30 ejecuciones realizadas, dependiendo del número de generación. También ofreceremos en la mayoría de las tablas de este capítulo la media de todas las medias de makespan, como un valor más que permite comparar las distintas configuraciones. En principio, la media de medias es un valor que no parece muy útil porque no representa nada concreto, pero en este caso en el que todas las medias de makespan tienen valores similares es algo más representativo para comparar varias configuraciones.

En la tabla se puede ver que hasta la generación número 300 se obtienen mejoras significativas en muchas de las instancias, en particular en las instancias ABZ y en las instancias t2-ps. En cambio, desde la generación 300 hasta la generación 600 las mejoras son ya muy pequeñas, y vemos que la media de todas las medias de makespan baja únicamente desde 1402.8 hasta 1402.1. Aunque observamos que entre la generación 300 y la generación 400 la media de makespan todavía baja una unidad en algunas de las instancias.

Tabla 8.1: Experimentos makespan: Ajuste inicial de parámetros

	Número de generación					
Instancia	100	200	300	400	500	600
t2-ps11	1448.6	1442.3	1440.4	1440.1	1440.0	1440.0
t2-ps12	1330.5	1320.2	1315.8	1315.0	1314.7	1313.8
t2-ps13	1427.5	1423.7	1422.3	1421.8	1421.3	1421.3
t2-ps14	1513.2	1501.9	1498.5	1497.3	1496.3	1496.0
t2-ps15	1517.8	1512.1	1510.4	1509.4	1509.4	1509.4
ABZ7sdst	1285.9	1258.4	1254.5	1254.5	1254.5	1254.5
ABZ8sdst	1329.7	1295.1	1289.8	1288.9	1288.6	1288.5
ABZ9sdst	1287.4	1255.9	1251.5	1250.2	1250.1	1250.1
LA21sdst	1291.3	1289.9	1289.9	1289.9	1289.9	1289.9
LA24sdst	1157.0	1156.3	1156.1	1156.1	1156.1	1156.1
LA25sdst	1196.0	1192.3	1192.1	1192.0	1192.0	1192.0
LA27sdst	1794.6	1778.8	1776.1	1776.1	1776.1	1775.9
LA29sdst	1683.5	1680.6	1680.5	1680.5	1680.5	1680.5
LA38sdst	1468.7	1465.2	1464.3	1464.3	1464.2	1464.1
LA40sdst	1502.7	1501.3	1500.5	1500.2	1500.1	1499.9
Media	1415.6	1404.9	1402.8	1402.4	1402.3	1402.1
T(s)	23.0	35.1	42.2	45.7	50.5	53.7

En la tabla 8.1 también indicamos el tiempo medio de una única ejecución, dependiendo del número de generación. Observamos que las generaciones iniciales del genético consumen mucho más tiempo de proceso, debido a que la población todavía no es muy buena y por lo tanto la búsqueda local necesita muchas iteraciones para llegar a óptimos locales. Esto también lo podemos apreciar de forma gráfica en la figura 8.1.

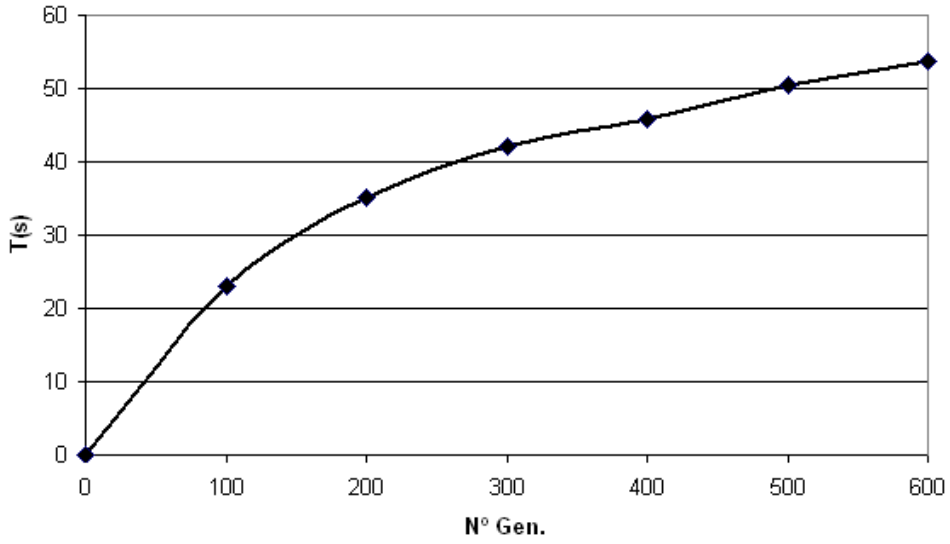


Figura 8.1: Experimentos makespan: Convergencia del algoritmo genético híbrido. Evolución del tiempo según el número de generación

Por otra parte, en las gráficas de la figura 8.2 podemos ver de forma gráfica la evolución del makespan medio respecto al número de generación del algoritmo genético híbrido, en cuatro de las instancias de prueba, aunque las gráficas son similares para las demás instancias. Podemos ver que por lo general la convergencia se obtiene entre la generación 200 y la generación 400.

A partir de los experimentos realizados en esta sección, decidimos fijar el tamaño de población en 200 y el número de generaciones en 400 para el método propuesto inicialmente en este estudio experimental. Aunque la mejora obtenida entre la generación 200 y la generación 400 no es muy elevada, consideramos que puede merecer la pena debido a que el aumento en tiempo de ejecución es reducido y en algunas instancias todavía se observan mejoras.

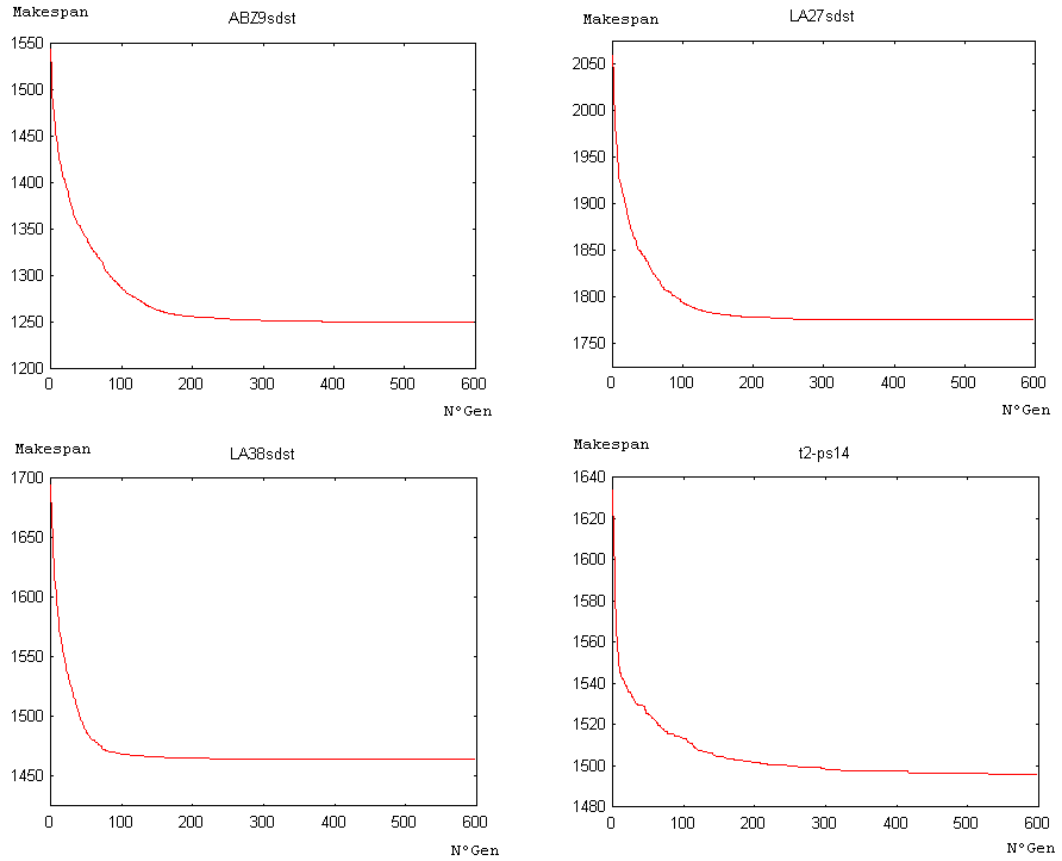


Figura 8.2: Experimentos makespan: Convergencia del algoritmo genético híbrido. Evolución del makespan según el número de generación

8.3.3. Comparación entre estructuras de vecindad

En la siguiente serie de experimentos evaluaremos las estructuras de vecindad propuestas en el capítulo 7: N_1^S y N^S. Para empezar vamos a comparar ambas en un algoritmo de búsqueda local básico como es la escalada, y después las compararemos en una combinación de algoritmo genético y búsqueda local. En ambos casos utilizamos un constructor estándar de planificaciones semiactivas. El objetivo de comparar las estructuras en un algoritmo básico de búsqueda local es mostrar que la ventaja de una estructura frente a la otra no es resultado de la combinación híbrida.

Al utilizar la búsqueda local de forma aislada el número de ejecuciones ha sido de 50000 para N_1^S y de 20000 para N^S. De esta forma el tiempo de ejecución es similar, situándose para las dos vecindades alrededor de 48 segundos de media por ejecución. La tabla 8.2 resume los resultados de estos experimentos para el conjunto de 15 instancias utilizado. En concreto

se muestra el mejor valor alcanzado en las 30 ejecuciones realizadas, el valor medio obtenido y la desviación estándar. Además, entre paréntesis al lado del mejor resultado se indica el número de veces que se ha conseguido dicho resultado.

Tabla 8.2: Experimentos makespan: Comparación entre las estructuras de vecindad N_1^S y N^S en una búsqueda local

	N_1^S			N^S		
Instancia	Mejor	Media	Desv	Mejor	Media	Desv
t2-ps11	1748(1)	1794.3	21.81	1546(2)	**1571.0**	15.33
t2-ps12	1567(5)	1589.0	15.02	1369(1)	**1400.8**	17.08
t2-ps13	1673(1)	1713.1	16.35	1470(1)	**1504.3**	14.80
t2-ps14	1686(1)	1730.9	15.61	1542(2)	**1566.7**	17.08
t2-ps15	1836(1)	1890.6	24.58	1597(1)	**1647.8**	17.07
ABZ7sdst	1541(1)	1620.6	22.46	1420(1)	**1451.8**	13.19
ABZ8sdst	1607(1)	1649.6	18.06	1433(1)	**1473.8**	14.96
ABZ9sdst	1607(2)	1648.8	19.12	1425(1)	**1470.7**	16.21
LA21sdst	1415(2)	1440.6	15.12	1351(2)	**1380.5**	13.35
LA24sdst	1257(2)	1286.5	13.44	1219(1)	**1241.3**	9.66
LA25sdst	1315(1)	1343.3	15.86	1237(1)	**1283.0**	15.15
LA27sdst	2133(1)	2181.1	20.38	1919(1)	**1983.7**	20.11
LA29sdst	2088(1)	2126.8	18.08	1854(1)	**1901.4**	17.62
LA38sdst	1635(1)	1680.7	15.18	1602(1)	**1627.9**	11.73
LA40sdst	1656(1)	1696.1	18.85	1610(1)	**1640.0**	13.97
Media		1692.8			**1543.0**	

En la tabla marcamos en negrita el mejor valor medio obtenido por las dos configuraciones. N^S obtiene en las 15 instancias un mejor resultado medio que N_1^S, siendo los tiempos de ejecución similares. Además, los mejores resultados alcanzados por N^S siempre son iguales o mejores que N_1^S. Por otra parte, la desviación estándar obtenida por las dos vecindades es en general comparable, aunque en 13 de las 15 instancias N^S obtiene desviaciones estándar más bajas, lo que significa que los resultados que ofrece son más estables.

A continuación proponemos una combinación de algoritmo genético con un método de búsqueda local de tipo máximo gradiente, utilizando de nuevo un constructor estándar de planificaciones semiactivas. Para conseguir tiempos de ejecución similares, el algoritmo genético se configura con valores diferentes dependiendo de la estructura de vecindad. Los parámetros (/tamaño de la población/número de generaciones/) han sido /340/680/ para N_1^S y /200/400/ para N^S. Con este ajuste de parámetros, el tiempo medio de una única

ejecución se sitúa alrededor de 48 segundos para las dos vecindades, al igual que en anteriores experimentos. La tabla 8.3 resume los resultados de estos experimentos.

Tabla 8.3: Experimentos makespan: Comparación entre las estructuras de vecindad N_1^S y N^S en un algoritmo memético

	N_1^S			N^S		
Instancia	Mejor	Media	Desv	Mejor	Media	Desv
t2-ps11	1438(1)	1494.8	28.48	1438(19)	**1439.8**	2.66
t2-ps12	1274(1)	1352.5	21.60	1269(3)	**1303.3**	29.21
t2-ps13	1430(6)	1439.3	9.22	1415(14)	**1424.3**	9.35
t2-ps14	1532(1)	1547.3	10.08	1492(26)	**1497.3**	13.83
t2-ps15	1503(1)	1540.6	14.21	1485(1)	**1508.0**	9.82
ABZ7sdst	1248(1)	1302.1	21.68	1233(2)	**1254.8**	12.90
ABZ8sdst	1282(1)	1330.7	22.12	1254(1)	**1286.4**	11.22
ABZ9sdst	1261(1)	1286.6	20.68	1217(1)	**1246.4**	10.73
LA21sdst	1286(8)	1291.3	4.66	1282(1)	**1291.1**	5.02
LA24sdst	1154(15)	1156.1	4.02	1151(8)	**1155.2**	3.92
LA25sdst	1188(6)	1196.2	6.20	1183(1)	**1190.4**	4.80
LA27sdst	1785(1)	1814.1	18.95	1761(1)	**1775.9**	8.88
LA29sdst	1678(3)	1701.0	17.08	1672(1)	**1681.5**	4.13
LA38sdst	1446(1)	1472.2	12.60	1446(2)	**1465.2**	12.93
LA40sdst	1493(2)	1502.6	5.80	1491(1)	**1499.6**	5.28
Media		1428.5			**1401.3**	

De nuevo N^S obtiene en las 15 instancias un mejor resultado medio que N_1^S a igualdad de tiempos de ejecución. Además, los mejores resultados alcanzados por N^S son siempre iguales o mejores a los alcanzados por N_1^S. En cuanto a la desviación estándar, podemos ver que en las instancias de mayor tamaño N^S consigue desviaciones estándar más bajas, por lo que sus resultados son más estables en general que los de N_1^S.

También hemos realizado experimentos para comparar N_1^S con la extensión trivial de N_1 que considera únicamente movimientos en los bordes de los bloques críticos, tal y como hace N_1 en el JSP sin tiempos de setup. Los experimentos se realizaron ejecutando el algoritmo memético por un tiempo 15 % superior al tiempo dado a N_1^S. En todas las instancias de prueba las mejores soluciones y las soluciones medias han sido peores con la extensión trivial.

Por todos estos motivos, la estructura más eficiente en esta serie de experimentos ha resultado ser la más compleja en el sentido de que es la que más vecinos genera: la estructura N^S propuesta en esta tesis. Obtiene mejores resultados incluso aunque su mayor complejidad

le permita utilizar un menor número de individuos en la población y un menor número de generaciones. Por tanto, en el resto de la experimentación sobre minimización de funciones objetivo de tipo bottleneck utilizaremos la estructura de vecindad N^S. Para las funciones objetivo de tipo suma realizaremos otra comparativa entre las vecindades, porque las conclusiones podrían ser diferentes. Esto es debido a que el alto número de caminos críticos existente en ese tipo de funciones quizás haga más recomendable la estructura de vecindad más simple, para que el número de vecinos no sea excesivo.

8.3.4. Comparación entre algoritmos de decodificación

En esta sección veremos la influencia que tiene el constructor de planificaciones utilizado. Lo estudiaremos en tres contextos diferentes: en el algoritmo memético, en un algoritmo genético simple y en una búsqueda local de tipo máximo gradiente. Empezaremos con el algoritmo memético, y en este caso recordemos que el planificador se utiliza únicamente en el momento en que se genera un individuo nuevo: bien en la población inicial o bien después de cada cruce, con el propósito de aplicar la búsqueda local a la planificación resultante. El constructor no se aplica al vecino elegido en cada iteración de la búsqueda local, ya que éste se generará simplemente cambiando determinados arcos en el grafo solución, tal y como se explica en la sección 7.7.1. Compararemos los constructores de planificaciones definidos en la sección 3.3, en concreto un constructor de planificaciones semiactivas, el constructor de planificaciones activas $EGYT1$ y el constructor de planificaciones activas $SSGS$. En este caso hemos utilizado la misma configuración del algoritmo genético y la búsqueda local para los tres constructores: /200/400/, y la estructura de vecindad utilizada es N^S. La tabla 8.4 muestra los resultados del experimento, en concreto el mejor resultado y el resultado medio para los tres constructores. Los resultados mostrados para el planificador semiactivo son los mismos que ya aparecían en la sección anterior. En cuanto a los tiempos de ejecución empleados, debemos tener en cuenta que la ejecución con el planificador semiactivo ha consumido aproximadamente el 70 % del tiempo de ejecución empleado por los planificadores activos. En la tabla hemos marcado en negrita el mejor valor medio obtenido por las tres configuraciones.

Recordemos que $EGYT1$ no es dominante, como explicamos en la sección 3.3.1, mientras que SSGS sí que lo es. Si comparamos el planificador $EGYT1$ con $SSGS$, el primero sólo obtiene un valor medio ligeramente mejor en dos instancias, mientras que en las restantes instancias obtiene medias por lo general sensiblemente peores que $SSGS$. En cuanto a las

Tabla 8.4: Experimentos makespan: Comparación entre los diferentes constructores de planificaciones en un algoritmo memético

	Semiactivo		*EGYT1*		*SSGS*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(19)	**1439.8**	1438(7)	1447.8	1438(12)	1441.7
t2-ps12	1269(3)	1303.3	1269(4)	1304.5	1269(15)	**1282.1**
t2-ps13	1415(14)	1424.3	1415(21)	**1419.6**	1415(21)	1420.1
t2-ps14	1492(26)	1497.3	1492(30)	**1492.0**	1492(30)	**1492.0**
t2-ps15	1485(1)	**1508.0**	1485(1)	1515.6	1485(1)	1509.9
ABZ7sdst	1233(2)	**1254.8**	1258(1)	1281.8	1243(1)	1259.5
ABZ8sdst	1254(1)	**1286.4**	1284(1)	1315.2	1268(1)	1291.0
ABZ9sdst	1217(1)	**1246.4**	1260(1)	1281.4	1223(1)	1258.3
LA21sdst	1282(1)	1291.1	1290(16)	1291.5	1277(1)	**1291.0**
LA24sdst	1151(8)	1155.2	1151(6)	1156.5	1151(9)	**1154.9**
LA25sdst	1183(1)	1190.4	1183(2)	**1189.3**	1183(2)	1190.9
LA27sdst	1761(1)	1775.9	1764(1)	1782.1	1758(1)	**1774.6**
LA29sdst	1672(1)	1681.5	1664(2)	1678.1	1656(1)	**1675.6**
LA38sdst	1446(2)	**1465.2**	1462(1)	1475.6	1464(9)	1472.7
LA40sdst	1491(1)	1499.6	1491(1)	1501.7	1484(1)	**1497.5**
Media		1401.3		1408.8		**1400.8**

mejores soluciones alcanzadas, $EGYT1$ llega a una mejor en una instancia, a la misma solución en siete y a una peor en las restantes siete instancias. Entonces, a la vista de estos resultados, si se quiere utilizar un constructor de planificaciones activas es beneficioso que dicho constructor sea dominante, por lo que en adelante elegiremos $SSGS$ como constructor de planificaciones activas.

Por otra parte si comparamos el planificador semiactivo con $SSGS$, vemos que en las tres instancias más grandes (ABZ7sdst, ABZ8sdst y ABZ9sdst) los resultados de utilizar planificaciones exclusivamente semiactivas son mejores, mientras que en el resto de instancias los resultados son comparables. Además, ya hemos comentado que el tiempo de ejecución consumido por el planificador semiactivo es inferior al consumido por los otros. Por estos motivos consideramos que en esta serie de experimentos el mejor planificador es el semiactivo.

Sin embargo, puede haber otras situaciones en las que utilizar un constructor de planificaciones activas puede ser muy beneficioso. Vamos a analizar a continuación la idoneidad de un generador activo o semiactivo cuando se utilizan por separado la búsqueda local o el algoritmo genético.

Comparación de planificadores en la búsqueda local

A continuación estudiamos la influencia del planificador al utilizar una búsqueda local por separado, sin combinar con un algoritmo genético. En estos casos el constructor se aplica únicamente en la generación de las soluciones iniciales. La tabla 8.5 muestra que la mejor elección es utilizar un constructor de planificaciones activas, ya que en todas las instancias *SSGS* obtiene mejores resultados. Esta diferencia radica en que el makespan medio de una planificación activa aleatoria es mucho menor que el de una planificación semiactiva aleatoria. Por ello, en el caso de comenzar desde una solución activa posiblemente se llegará mucho antes a un óptimo local y se ahorrará gran parte del tiempo de ejecución.

Tabla 8.5: Experimentos makespan: Comparación entre los diferentes constructores de planificaciones en una búsqueda local

	Semiactivo		*SSGS*	
Instancia	Mejor	Media	Mejor	Media
t2-ps11	1546(2)	1571.0	1515(1)	**1555.1**
t2-ps12	1369(1)	1400.8	1350(1)	**1386.6**
t2-ps13	1470(1)	1504.3	1461(1)	**1500.3**
t2-ps14	1542(2)	1566.7	1542(4)	**1554.0**
t2-ps15	1597(1)	1647.8	1604(1)	**1637.2**
ABZ7sdst	1420(1)	1451.8	1395(1)	**1424.5**
ABZ8sdst	1433(1)	1473.8	1417(1)	**1453.3**
ABZ9sdst	1425(1)	1470.7	1402(1)	**1455.3**
LA21sdst	1351(2)	1380.5	1339(1)	**1367.3**
LA24sdst	1219(1)	1241.3	1192(1)	**1230.6**
LA25sdst	1237(1)	1283.0	1251(1)	**1270.3**
LA27sdst	1919(1)	1983.7	1913(1)	**1964.7**
LA29sdst	1854(1)	1901.4	1815(1)	**1875.4**
LA38sdst	1602(1)	1627.9	1571(1)	**1601.0**
LA40sdst	1610(1)	1640.0	1574(1)	**1609.6**
Media		1543.0		**1525.7**

Hemos ajustado los parámetros pensando en la comparación con el algoritmo genético híbrido, para que las ejecuciones consuman aproximadamente tiempos similares a los utilizados en experimentos anteriores. En el caso de planificaciones semiactivas se han realizado 20000 búsquedas locales y en el caso del constructor *SSGS* se han realizado 35000 búsquedas locales, siendo en ambos casos el tiempo medio de una única ejecución de 48 segundos, al igual que en los experimentos realizados anteriormente. Entonces, el hecho utilizar un

constructor de planificaciones activas ha permitido que en cada búsqueda local el tiempo de ejecución sea menor, y eso permite realizar más búsquedas locales en el mismo tiempo y obtener un mejor resultado final.

Comparación de planificadores en el algoritmo genético simple

Por último, estudiaremos las diferencias entre los planificadores en el caso de utilizar el algoritmo genético por separado. Al igual que en el algoritmo memético, los planificadores se aplican para generar la población inicial, y a los individuos resultantes de cada operación de cruce, con el propósito de calcular su función objetivo. La tabla 8.6 muestra los resultados de dos experimentos, utilizando en uno de ellos un algoritmo genético con un constructor de planificaciones semiactivo, y en el otro un constructor activo (elegimos SSGS). En los dos casos los parámetros del algoritmo genético son /1000/1200/. Hemos ajustado los parámetros para que las ejecuciones con el planificador semiactivo consuman aproximadamente el mismo tiempo de ejecución que en anteriores experimentos, aunque las ejecuciones con SSGS han consumido un tiempo mayor.

Tabla 8.6: Experimentos makespan: Comparación entre los diferentes constructores de planificaciones en un algoritmo genético

	Semiactivo			*SSGS*		
Instancia	Mejor	Media	Desv	Mejor	Media	Desv
t2-ps11	1519(1)	**1611.3**	41.58	1567(1)	1633.8	28.12
t2-ps12	1357(1)	**1460.7**	39.95	1378(1)	1462.9	26.23
t2-ps13	1500(2)	**1549.5**	32.77	1500(1)	1561.5	35.45
t2-ps14	1552(2)	**1610.6**	33.08	1552(1)	1624.4	25.65
t2-ps15	1588(1)	**1647.6**	34.02	1618(1)	1686.5	25.92
ABZ7sdst	1447(1)	**1550.7**	47.17	1532(1)	1579.0	17.81
ABZ8sdst	1475(1)	**1585.9**	52.26	1561(1)	1596.9	16.84
ABZ9sdst	1438(1)	**1536.3**	41.98	1519(1)	1558.2	22.95
LA21sdst	1321(1)	1349.9	15.14	1323(1)	**1345.2**	13.75
LA24sdst	1154(1)	**1204.0**	20.77	1196(1)	1226.4	15.89
LA25sdst	1229(2)	1259.0	18.23	1225(1)	**1253.5**	12.00
LA27sdst	1925(1)	**2002.2**	38.85	2028(2)	2068.5	24.84
LA29sdst	1829(1)	**1910.6**	48.96	1920(1)	1972.4	27.89
LA38sdst	1525(1)	1573.9	24.51	1528(2)	**1544.2**	11.17
LA40sdst	1558(1)	1591.6	20.35	1557(1)	**1582.6**	13.74
Media		**1562.9**			1579.7	

Sin embargo en la tabla observamos que en todos los casos el planificador semiactivo obtiene mejores resultados, a pesar de utilizar un tiempo de ejecución menor. Por lo tanto en esta serie de experimentos es claramente mejor el planificador semiactivo. Aunque también debemos observar que las desviaciones estándar en el caso del constructor semiactivo son bastante mayores que en el caso del constructor activo, lo que indica una mayor variabilidad en los resultados.

Pero como veremos ahora, la elección del constructor de planificaciones también depende en gran medida del tiempo de ejecución utilizado. Las gráficas 8.3 muestran la evolución del makespan medio respecto al número de generación del algoritmo genético, en cuatro de las instancias de prueba.

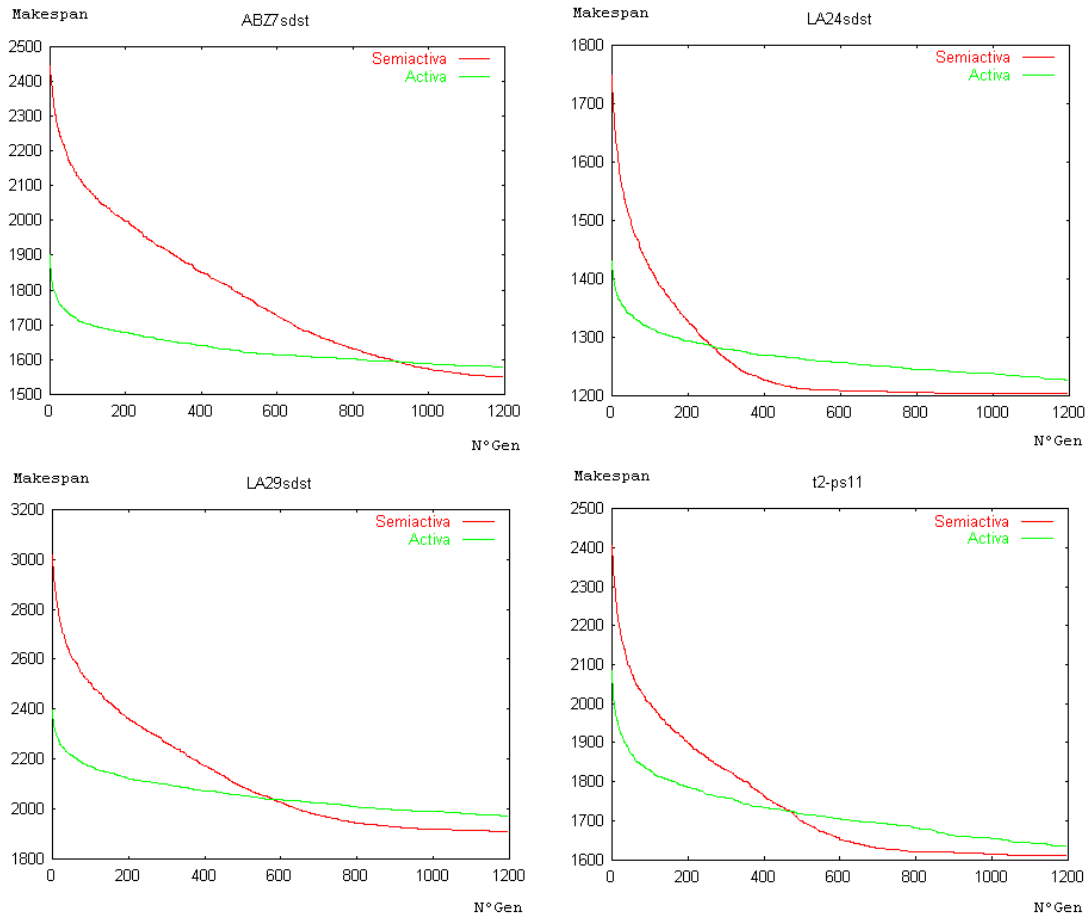


Figura 8.3: Experimentos makespan: Planificaciones semiactivas frente a activas en un algoritmo genético. Evolución del makespan según el número de generación

La línea roja representa la evolución con el planificador semiactivo y la línea verde la evolución con *SSGS*. Podemos observar que *SSGS* siempre comienza con makespan medios mucho más bajos, lo cual es lógico dado que una planificación activa aleatoria es mejor que una planificación semiactiva aleatoria. Sin embargo, si el tiempo de ejecución es lo suficientemente elevado, vemos que las planificaciones semiactivas tienen una mejor capacidad de evolucionar y se acaban consiguiendo mejores resultados, probablemente debido a que reducir el espacio de búsqueda puede limitar el número de soluciones visitadas y quizás causar una convergencia prematura. Por lo tanto, a la vista de las gráficas parece ser que utilizando tiempos de ejecución muy reducidos es aconsejable utilizar planificaciones activas en el genético, pero con tiempos de ejecución elevados es preferible no reducir el espacio de búsqueda y utilizar planificaciones semiactivas. En concreto, en la instancia ABZ7sdst las planificaciones semiactivas comenzaron a obtener mejores resultados a partir de la generación 900 aproximadamente, mientras que en la instancia LA29sdst a partir de la 600 aproximadamente, en la t2-ps11 en la generación 500, y en la LA24sdst alrededor de la 250. Aunque sólo mostramos como ejemplo el gráfico de cuatro de las instancias, este mismo comportamiento se ha observado en la mayoría de las 15 instancias de prueba.

8.3.5. Comparación entre los métodos por separado y combinados

Hemos comenzado este estudio experimental con un híbrido de algoritmo genético y búsqueda local de tipo máximo gradiente. En la sección anterior también hemos mostrado resultados sobre cada uno de los dos métodos por separado para elegir el planificador más adecuado en cada caso. En esta sección mostramos en la tabla 8.7 los mejores resultados obtenidos por cada método resumidos en una misma tabla, para mayor claridad y facilidad de comparación. Los parámetros están elegidos para obtener tiempos de ejecución similares con todos los métodos, y son los indicados en la sección anterior. Los tiempos de ejecución se sitúan alrededor de 48 segundos por ejecución. Recordemos que los algoritmos se codifican en C++ y se ejecutan en un Intel Core 2 Duo a 2.66GHz con 2Gb de RAM.

Como se puede observar en los mejores valores y los valores medios de la tabla 8.7, el algoritmo genético por separado, y la búsqueda local por separado utilizando N^S obtienen resultados similares, aunque por lo general es mejor la búsqueda local, ya que en 10 de las 15 instancias obtiene una media de makespan más baja que el algoritmo genético. Sin embargo, cualquiera de los dos métodos por separado es claramente inferior en todas las instancias a la combinación de ambos, utilizando también la estructura N^S.

Por otra parte, la media de la desviación estándar de las mejores soluciones alcanzadas en las 30 ejecuciones en los algoritmos de búsqueda local es menor que la de los algoritmos genéticos. Estas diferencias indican que los valores medios obtenidos mediante los algoritmos de búsqueda local son ligeramente más representativos que los valores de los algoritmos genéticos.

Tabla 8.7: Experimentos makespan: Comparación entre los métodos por separado o combinados

	BL			*AG*			*AG+BL*		
Instancia	Mejor	Media	Desv	Mejor	Media	Desv	Mejor	Media	Desv
t2-ps11	1515(1)	1555.1	19.10	1519(1)	1611.3	41.58	1438(19)	**1439.8**	2.66
t2-ps12	1350(1)	1386.6	17.02	1357(1)	1460.7	39.95	1269(3)	**1303.3**	29.21
t2-ps13	1461(1)	1500.3	13.30	1500(2)	1549.5	32.77	1415(14)	**1424.3**	9.35
t2-ps14	1542(4)	1554.0	8.82	1552(2)	1610.6	33.08	1492(26)	**1497.3**	13.83
t2-ps15	1604(1)	1637.2	15.48	1588(1)	1647.6	34.02	1485(1)	**1508.0**	9.82
ABZ7sdst	1395(1)	1424.5	13.92	1447(1)	1550.7	47.17	1233(2)	**1254.8**	12.90
ABZ8sdst	1417(1)	1453.3	13.07	1475(1)	1585.9	52.26	1254(1)	**1286.4**	11.22
ABZ9sdst	1402(1)	1455.3	16.84	1438(1)	1536.3	41.98	1217(1)	**1246.4**	10.73
LA21sdst	1339(1)	1367.3	11.83	1321(1)	1349.9	15.14	1282(1)	**1291.1**	5.02
LA24sdst	1192(1)	1230.6	13.72	1154(1)	1204.0	20.77	1151(8)	**1155.2**	3.92
LA25sdst	1251(1)	1270.3	7.64	1229(2)	1259.0	18.23	1183(1)	**1190.4**	4.80
LA27sdst	1913(1)	1964.7	16.84	1925(1)	2002.2	38.85	1761(1)	**1775.9**	8.88
LA29sdst	1815(1)	1875.4	18.85	1829(1)	1910.6	48.96	1672(1)	**1681.5**	4.13
LA38sdst	1571(1)	1601.0	9.75	1525(1)	1573.9	24.51	1446(2)	**1465.2**	12.93
LA40sdst	1574(1)	1609.6	13.97	1558(1)	1591.6	20.35	1491(1)	**1499.6**	5.28
Media		1525.7			1562.9			**1401.3**	

Además, a partir de la desviación estándar podemos construir el intervalo de confianza al 95 % para la media del makespan obtenida por los distintos métodos. Para empezar, calculamos el error estándar (SE) de la media dividiendo la desviación estándar (σ) por la raíz cuadrada del tamaño muestral, es decir $SE = \sigma/\sqrt{30}$. Ahora podemos calcular el intervalo de confianza: el límite inferior será la media (μ) menos 1.96 veces el error estándar de la media y el límite superior la media (μ) más 1.96 veces el error estándar de la media, es decir, $\mu \pm 1{,}96 * SE$. Haciendo los cálculos observamos que el extremo superior del intervalo de confianza del algoritmo genético híbrido es mucho menor que el extremo inferior de la búsqueda local y que el extremo inferior del algoritmo genético simple. En resumen, la eficiencia de ambos métodos es similar cuando se utilizan por separado, pero combinándolos

los resultados mejoran considerablemente.

8.3.6. Escalada de máximo gradiente frente a escalada simple

Hasta este momento hemos utilizado una estrategia de búsqueda local de tipo escalada por máximo gradiente, es decir que se genera toda la vecindad y se estima el makespan de todos los vecinos, y sólo después de todo este proceso se elige al mejor de todos ellos. Una estrategia alternativa de escalada consiste en ir generando los vecinos uno por uno, y en cuanto se encuentre un vecino cuya estimación del makespan sea menor al makespan del individuo original, se elige inmediatamente ese vecino. Este método alternativo se denomina *escalada simple*. Intuitivamente este método ahorraría mucho tiempo de ejecución ya que normalmente no necesitará generar toda la vecindad, aunque por otra parte debemos pensar que consumirá más tiempo de ejecución porque, al ser más pequeñas las mejoras entre individuos sucesivos, se necesitará un mayor número de iteraciones para alcanzar un óptimo local.

Esta sección muestra experimentos comparando ambas técnicas de escalada, y además en el caso de la escalada simple probaremos tanto un constructor de planificaciones semiactivo como *SSGS*. En este caso pensamos que la utilización de *SSGS* puede ser beneficiosa ya que en la escalada simple el proceso de mejora de los individuos es mucho más lento, y el hecho de comenzar desde una solución inicial mejor puede dar como resultado una elevada diferencia en tiempo de ejecución.

La tabla 8.8 muestra los resultados de esta serie de experimentos. Todos ellos se ejecutan con la misma configuración utilizada anteriormente, es decir /200/400/, excepto en el caso de los problemas ABZ y LA para el máximo gradiente. Para estas 10 instancias el método de máximo gradiente consume un tiempo de ejecución considerablemente menor a la escalada simple, por lo tanto para mantener una cierta igualdad de tiempos hemos optado por repetir los experimentos de esas 10 instancias con una configuración de /300/600/ en el caso del máximo gradiente. En esta serie de experimentos hemos utilizado un tiempo de ejecución medio de unos 80 segundos por prueba, para las tres configuraciones.

Como viene siendo habitual hemos marcado en negrita en cada instancia el método que ha alcanzado un mejor resultado medio. Podemos observar que los tres métodos obtienen resultados similares. Consideramos como mejor método en esta serie de experimentos a la combinación de escalada simple con planificaciones semiactivas, aunque la diferencia con las demás configuraciones no es elevada. Un dato llamativo es que la diferencia entre utilizar

Tabla 8.8: Experimentos makespan: Utilización de diferentes tipos de escalada

	Máximo gradiente Semiactiva		*Escalada simple Semiactiva*		*Escalada simple SSGS*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(19)	1439.8	1438(30)	**1438.0**	1438(21)	1439.4
t2-ps12	1269(3)	1303.3	1269(2)	1305.0	1269(2)	**1295.3**
t2-ps13	1415(14)	1424.3	1410(1)	**1414.8**	1415(28)	1416.0
t2-ps14	1492(26)	1497.3	1479(2)	1491.1	1452(1)	**1488.6**
t2-ps15	1485(1)	1508.0	1487(2)	1509.9	1485(2)	**1505.3**
ABZ7sdst	1233(1)	1250.1	1227(1)	**1248.3**	1239(1)	1259.9
ABZ8sdst	1266(1)	1282.1	1259(1)	**1275.9**	1266(1)	1286.3
ABZ9sdst	1217(1)	1242.3	1217(2)	**1239.0**	1220(1)	1255.1
LA21sdst	1282(2)	1289.5	1277(1)	**1288.9**	1277(2)	1290.5
LA24sdst	1151(8)	**1154.1**	1151(3)	1155.9	1151(7)	1156.0
LA25sdst	1183(1)	**1188.5**	1188(16)	1191.7	1183(1)	1192.7
LA27sdst	1760(1)	**1771.5**	1758(2)	1772.7	1760(1)	1774.9
LA29sdst	1677(3)	1681.3	1664(1)	1676.5	1664(4)	**1675.0**
LA38sdst	1446(3)	1460.9	1446(13)	**1454.2**	1456(1)	1469.3
LA40sdst	1488(1)	1497.7	1489(1)	1498.4	1492(1)	**1497.0**
Media		1399.4		**1397.4**		1400.1

planificaciones activas o semiactivas es bastante grande al considerar el algoritmo genético o la búsqueda local por separado, pero las diferencias se reducen considerablemente al combinar los dos métodos.

8.3.7. Utilización de búsqueda tabú

A continuación introducimos la búsqueda tabú, para comprobar si es capaz de mejorar los resultados obtenidos en los estudios experimentales realizados anteriormente. La búsqueda tabú utilizada es la descrita en la sección 6.4. Una de las ventajas del algoritmo descrito es que no requiere una calibración previa de los parámetros. Por ejemplo el número máximo de iteraciones *maxGlobalIter* dependerá principalmente del tiempo de ejecución deseado. La idea es comparar el mejor método obtenido en los anteriores estudios experimentales con otros dos métodos: la utilización de una búsqueda tabú por separado, y un híbrido de algoritmo genético combinado con búsqueda tabú. Además, compararemos estos métodos en dos tipos de ejecuciones: una con el tiempo de ejecución que veníamos utilizando en las últimas secciones, y otra con un tiempo de ejecución mucho más reducido, debido a que

quizás un método pueda ser mejor que otro dependiendo del tiempo disponible.

Calibración del peso de la búsqueda tabú en el algoritmo híbrido

Al utilizar la búsqueda tabú en combinación con un algoritmo genético, surge la duda de cuánto peso debemos dar a la búsqueda tabú respecto al genético. Para mantener la igualdad de tiempos, es claro que si aumentamos el número de iteraciones máximas de la búsqueda tabú debemos reducir los parámetros del algoritmo genético. La tabla 8.9 muestra experimentos utilizando diferentes valores de iteraciones máximas para la búsqueda tabú (en concreto elegimos los valores 50, 200 y 800). Los experimentos realizados en esta sección han consumido alrededor de 66 segundos de media por prueba. Para mantener esta igualdad de tiempos hemos tenido que utilizar diferentes configuraciones dependiendo del tamaño de la instancia. En el caso intermedio (200 iteraciones de búsqueda tabú) los parámetros utilizados (/ tamaño de la población / número de generaciones / iteraciones de búsqueda tabú /) son:

- /60/85/200/ para las instancias LA27sdst y LA29sdst.
- /80/110/200/ para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst.
- /40/60/200/ para todas las restantes instancias.

En el caso de dar más peso al algoritmo genético los parámetros son:

- /110/155/50/ para las instancias LA27sdst y LA29sdst.
- /140/200/50/ para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst.
- /80/110/50/ para todas las restantes instancias.

Por último, en el caso de dar más peso a la búsqueda tabú, los parámetros son:

- /30/45/800/ para las instancias LA27sdst y LA29sdst.
- /40/60/800/ para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst.
- /20/30/800/ para todas las restantes instancias.

Como es natural, debemos cambiar el tamaño de población y el número de generaciones para mantener el tiempo de ejecución cuando cambiamos el número de iteraciones máximas de la búsqueda tabú.

Tabla 8.9: Experimentos makespan: Peso de la búsqueda tabú respecto al algoritmo genético

	MaxIter=50		*MaxIter=200*		*MaxIter=800*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(22)	**1438.9**	1438(10)	1440.8	1438(2)	1444.3
t2-ps12	1269(12)	1277.8	1269(10)	**1277.1**	1272(1)	1280.8
t2-ps13	1415(27)	1416.1	1415(28)	1416.0	1415(30)	**1415.0**
t2-ps14	1478(1)	1488.9	1452(1)	**1488.8**	1492(30)	1492.0
t2-ps15	1485(8)	1497.9	1485(10)	**1495.9**	1485(1)	1499.9
ABZ7sdst	1227(1)	1249.3	1231(1)	**1245.3**	1235(1)	1249.1
ABZ8sdst	1263(2)	1280.2	1257(1)	**1271.2**	1257(1)	1275.8
ABZ9sdst	1220(1)	1246.2	1217(2)	**1234.6**	1219(1)	1242.9
LA21sdst	1273(5)	1284.2	1268(6)	1276.0	1268(8)	**1273.3**
LA24sdst	1151(14)	1153.4	1151(15)	**1152.6**	1151(8)	1153.9
LA25sdst	1183(6)	1188.0	1183(5)	1187.6	1183(9)	**1186.5**
LA27sdst	1738(1)	1759.8	1735(1)	**1753.7**	1734(1)	1755.3
LA29sdst	1659(4)	1670.2	1659(4)	**1666.7**	1651(2)	1669.6
LA38sdst	1446(6)	1460.2	1446(17)	**1451.7**	1446(4)	1454.7
LA40sdst	1488(1)	1494.4	1482(1)	**1489.7**	1483(1)	1490.7
Media		1393.7		**1389.8**		1392.3

Los resultados de la tabla 8.9 muestran que la mejor opción, para el tiempo de ejecución elegido, es utilizar un número de iteraciones máximas para la búsqueda tabú intermedio. En estos experimentos vemos que es la configuración que obtiene las mejores medias en 11 de las 15 instancias de prueba. Aunque debemos tener en cuenta que si disponemos de un tiempo de ejecución mayor o menor la mejor configuración podría cambiar. Hemos realizado también experimentos con iteraciones máximas de 100 y 400, y los resultados son similares a los obtenidos con 200 iteraciones máximas y en ambos casos algo mejores que los obtenidos con 50 y 800 iteraciones máximas.

Un detalle curioso que podemos observar es cómo los resultados para la instancia t2-ps11 empeoran a medida que se le da más peso a la búsqueda tabú. De hecho, si consultamos en anteriores secciones vemos que el genético con escalada simple y planificaciones semiactivas era capaz de obtener en esta instancia el óptimo en todas las ejecuciones. Por algún motivo la búsqueda tabú se comporta mal en esta instancia.

Comparación de los distintos métodos

La tabla 8.10 muestra los resultados de la comparación de los distintos métodos. En cuanto al híbrido de algoritmo genético y búsqueda tabú hemos elegido el que mejor resultado ha ofrecido en la comparación anterior, y por lo tanto los parámetros utilizados han sido de /60/85/200/ para las instancias LA27sdst y LA29sdst, de /80/110/200/ para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst, y de /40/60/200/ para todas las restantes instancias. En el caso de utilizar la búsqueda tabú por separado hemos utilizado una lista de soluciones elite de tamaño 20, y hemos ajustado el número de iteraciones máximas para obtener tiempos de ejecución similares: 1300000 iteraciones máximas para las instancias LA27sdst y LA29sdst, 2200000 para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst, y 500000 para todas las restantes instancias. En el algoritmo genético con búsqueda local de tipo escalada la configuración es de /200/400/ para todas las instancias. Con los parámetros elegidos, el tiempo consumido por los tres métodos es comparable, aunque es ligeramente inferior en los dos casos en los que esta implicada la búsqueda tabú. En esos dos casos se utiliza una media de 66 segundos por prueba, mientras que el genético con búsqueda local simple utiliza una media de 80 segundos por prueba. Esto remarca todavía más la diferencia entre los métodos.

Tabla 8.10: Experimentos makespan: Resultados de la búsqueda tabú

	AG+BL		*TS*		*AG+TS*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(30)	**1438.0**	1438(1)	1458.3	1438(10)	1440.8
t2-ps12	1269(2)	1305.0	1274(1)	1297.6	1269(10)	**1277.1**
t2-ps13	1410(1)	**1414.8**	1415(11)	1420.2	1415(28)	1416.0
t2-ps14	1479(2)	1491.1	1492(25)	1493.2	1452(1)	**1488.8**
t2-ps15	1487(2)	1509.9	1506(1)	1517.6	1485(10)	**1495.9**
ABZ7sdst	1227(1)	1248.3	1262(1)	1273.3	1231(1)	**1245.3**
ABZ8sdst	1259(1)	1275.9	1286(1)	1300.6	1257(1)	**1271.2**
ABZ9sdst	1217(2)	1239.0	1256(1)	1270.7	1217(2)	**1234.6**
LA21sdst	1277(1)	1288.9	1268(2)	**1272.6**	1268(6)	1276.0
LA24sdst	1151(3)	1155.9	1153(1)	1155.4	1151(15)	**1152.6**
LA25sdst	1188(16)	1191.7	1183(4)	**1187.6**	1183(5)	**1187.6**
LA27sdst	1758(2)	1772.7	1750(1)	1773.2	1735(1)	**1753.7**
LA29sdst	1664(1)	1676.5	1668(1)	1686.4	1659(4)	**1666.7**
LA38sdst	1446(13)	1454.2	1455(1)	1465.9	1446(17)	**1451.7**
LA40sdst	1489(1)	1498.4	1481(1)	1491.2	1482(1)	**1489.7**
Media		1397.4		1404.3		**1389.8**

Como se puede observar en la tabla 8.10, la mayoría de las mejores medias se consiguen con el híbrido de algoritmo genético y búsqueda tabú, y en las pocas instancias en las que otro método consigue la mejor media, la diferencia respecto de éste es mínima. Por lo que concluímos que con un tiempo de ejecución lo suficientemente elevado, AG+TS es la mejor combinación de metaheurísticas. No hemos indicado las desviaciones estándar en la tabla porque eran similares en los tres casos y era difícil extraer ninguna conclusión sobre ellas.

Por otra parte debemos señalar que, según los resultados de otro experimento realizado, en los casos en los que se utiliza búsqueda tabú emplear el constructor semiactivo o *SSGS* produce resultados prácticamente idénticos, lo que confirma la tendencía que ya veníamos observando de que las diferencias entre planificaciones activas y semiactivas se van reduciendo a medida que aumenta la complejidad de las metaheurísticas utilizadas.

Comparación de los distintos métodos en ejecuciones cortas

En la siguiente serie de experimentos comparamos los mismos métodos en ejecuciones cortas. Arbitrariamente vamos a considerar una ejecución como corta si consume un tiempo cincuenta veces menor que el utilizado en anteriores experimentos. Para conseguir este tiempo, fijamos en 22000 el número de iteraciones máximas de la búsqueda tabú, mientras que la configuración del genético con escalada es de /30/50/ y la del genético con búsqueda tabú es /10/20/100/. Con estos parámetros, la duración media de una única ejecución, para los tres métodos, se sitúa alrededor de un segundo y medio. En este caso, al utilizar el genético con una búsqueda local simple hemos utilizado escalada de máximo gradiente en lugar de escalada simple ya que, como hemos constatado con experimentos adicionales, con un tiempo de ejecución muy reducido se comporta mejor. La tabla 8.11 muestra los resultados de estos experimentos.

Observamos que en este caso es la búsqueda tabú por separado el método que consigue la mayoría de mejores valores medios. Aunque en las instancias del conjunto BT no es del todo claro qué método es el mejor, en el resto de instancias es claramente mejor utilizar la búsqueda tabú por separado. De nuevo, no hemos indicado las desviaciones estándar porque eran muy similares en los tres casos.

Conclusiones sobre el uso de la búsqueda tabú

A partir de los resultados mostrados en las tablas 8.10 y 8.11 concluímos que la utilización de una búsqueda tabú por separado es el método más interesante cuando el tiempo de

Tabla 8.11: Experimentos makespan: Resultados de la búsqueda tabú en ejecuciones cortas

	AG+BL		*TS*		*AG+TS*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1449(1)	1504.0	1464(2)	1489.8	1444(2)	**1473.3**
t2-ps12	1328(1)	1350.1	1299(1)	**1331.7**	1275(1)	1332.1
t2-ps13	1430(2)	1442.6	1415(2)	1432.8	1415(2)	**1431.7**
t2-ps14	1492(2)	1533.0	1492(9)	**1509.1**	1492(12)	1512.7
t2-ps15	1529(2)	1558.3	1511(1)	1535.4	1516(2)	**1528.5**
ABZ7sdst	1305(1)	1350.5	1280(1)	**1305.0**	1297(1)	1330.7
ABZ8sdst	1350(1)	1384.1	1311(1)	**1337.3**	1318(1)	1348.4
ABZ9sdst	1322(1)	1369.7	1273(1)	**1316.1**	1295(1)	1336.1
LA21sdst	1290(1)	1314.9	1268(2)	**1288.2**	1273(1)	1297.6
LA24sdst	1163(1)	1183.5	1154(2)	1169.8	1151(1)	**1168.1**
LA25sdst	1195(1)	1211.2	1183(1)	**1197.3**	1188(1)	1202.3
LA27sdst	1813(1)	1859.9	1776(1)	**1814.9**	1787(1)	1820.2
LA29sdst	1693(1)	1757.4	1689(1)	**1715.9**	1689(1)	1720.7
LA38sdst	1495(2)	1518.8	1466(1)	**1494.2**	1470(2)	1495.2
LA40sdst	1506(1)	1530.0	1493(2)	**1506.0**	1495(1)	1508.7
Media		1457.9		**1429.6**		1433.8

ejecución disponible es muy reducido. Sin embargo cuando el tiempo de ejecución disponible es mayor empieza a ser beneficiosa la utilización de un híbrido entre algoritmo genético y búsqueda tabú. También hemos comprobado experimentalmente cuánto peso es más recomendable dar a cada metaheurística cuando se combinan. En vista de todo esto, para el resto de experimentos de optimización de parámetros utilizaremos el híbrido de algoritmo genético y búsqueda tabú. Aunque en el capítulo 9, dedicado a la minimización del maximum lateness, utilizaremos la búsqueda tabú exclusivamente debido a que los algoritmos con los que nos compararemos en dicho capítulo utilizan un tiempo de ejecución muy reducido.

8.3.8. Análisis de la estrategia evolutiva

En la sección 5.13 discutimos las dos formas clásicas de considerar la evolución de los individuos en un algoritmo memético: evolución Baldwiniana y evolución Lamarckiana. Para comparar estas opciones experimentalmente hemos realizado las pruebas cuyos resultados se muestran en la tabla 8.12. La configuración para estos experimentos es la misma que la utilizada en anteriores secciones. Las ejecuciones realizadas utilizando la evolución Baldwiniana han consumido un tiempo de ejecución el 5 % mayor que las realizadas con la evolución

Lamarckiana. Este hecho, unido a que los resultados son sensiblemente peores en todas las instancias, nos llevan a la conclución de que en la metaheurística que utilizamos lo más aconsejable es la evolución Lamarckiana, que es la que veníamos utilizando en todos los experimentos anteriores.

Tabla 8.12: Experimentos makespan: Comparación entre distintos tipos de evolución

	Evolución Lamarckiana		*Evolución Baldwiniana*	
Instancia	Mejor	Media	Mejor	Media
t2-ps11	1438(10)	**1440.8**	1444(2)	1466.7
t2-ps12	1269(10)	**1277.1**	1276(1)	1321.9
t2-ps13	1415(28)	**1416.0**	1415(4)	1426.3
t2-ps14	1452(1)	**1488.8**	1492(7)	1502.3
t2-ps15	1485(10)	**1495.9**	1512(1)	1526.3
ABZ7sdst	1231(1)	**1245.3**	1266(1)	1299.5
ABZ8sdst	1257(1)	**1271.2**	1311(1)	1332.6
ABZ9sdst	1217(2)	**1234.6**	1255(1)	1289.3
LA21sdst	1268(6)	**1276.0**	1277(1)	1291.0
LA24sdst	1151(15)	**1152.6**	1154(4)	1159.3
LA25sdst	1183(5)	**1187.6**	1188(4)	1194.8
LA27sdst	1735(1)	**1753.7**	1781(1)	1802.3
LA29sdst	1659(4)	**1666.7**	1685(1)	1702.9
LA38sdst	1446(17)	**1451.7**	1458(1)	1484.7
LA40sdst	1482(1)	**1489.7**	1497(1)	1506.3
Media		**1389.8**		1420.4

8.3.9. Análisis de estrategias de uso de estimaciones

Hasta este momento hemos considerado únicamente la opción de estimar el makespan de toda la vecindad generada, y posteriormente elegir el individuo con la estimación más pequeña de toda la vecindad. Intuitivamente es la mejor opción ya que en la sección 7.6.5 hemos comprobado que el algoritmo de estimación del makespan es muy preciso. Sin embargo, dado que la estimación no es exacta, conviene considerar más posibilidades.

Una de ellas es comprobar de forma exacta el makespan de todos los individuos cuya estimación es menor que el makespan del individuo original, y de entre ellos quedarnos con el vecino de menor makespan. Los vecinos con una estimación mayor que el makespan del individuo original los descartamos porque en la sección 7.6.5 hemos visto que muy rara vez

la estimación es mayor que el makespan real del individuo.

Por último, también tenemos la opción, de no utilizar estimaciones y evaluar de forma exacta toda la vecindad, con lo cual también tendríamos en cuenta los individuos cuya estimación es mayor que el makespan del individuo original, pero que su makespan real es más reducido. Esta opción es intuitivamente la más desaconsejable ya que consumirá un tiempo de ejecución mucho mayor y no se espera que consiga una gran mejora respecto a las otras.

Para comparar estas tres opciones hemos realizado las pruebas cuyos resultados mostramos en la tabla 8.13. La configuración para los experimentos es la misma que la utilizada en anteriores secciones.

Tabla 8.13: Experimentos makespan: Diferentes formas de utilizar el algoritmo de estimación

	Sólo mejor estim.		*Estim. <Makespan*		*Sin estimaciones*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(10)	1440.8	1438(12)	1440.7	1438(13)	**1439.8**
t2-ps12	1269(10)	**1277.1**	1269(3)	1283.0	1269(2)	1282.8
t2-ps13	1415(28)	1416.0	1415(25)	1416.8	1415(27)	**1415.7**
t2-ps14	1452(1)	1488.8	1452(4)	1485.4	1452(4)	**1483.6**
t2-ps15	1485(10)	1495.9	1485(4)	1497.9	1485(5)	**1495.5**
ABZ7sdst	1231(1)	1245.3	1227(1)	**1243.7**	1228(1)	1246.2
ABZ8sdst	1257(1)	**1271.2**	1258(1)	1274.0	1265(1)	1276.2
ABZ9sdst	1217(2)	**1234.6**	1217(3)	1237.7	1219(1)	1236.6
LA21sdst	1268(6)	1276.0	1268(8)	**1275.0**	1268(11)	1276.4
LA24sdst	1151(15)	1152.6	1151(20)	**1152.4**	1151(19)	1152.8
LA25sdst	1183(5)	1187.6	1187(1)	1188.2	1183(6)	**1187.2**
LA27sdst	1735(1)	**1753.7**	1734(1)	1758.5	1735(1)	1761.4
LA29sdst	1659(4)	**1666.7**	1658(1)	1666.9	1659(1)	1666.8
LA38sdst	1446(17)	**1451.7**	1446(10)	1455.6	1446(11)	1454.5
LA40sdst	1482(1)	**1489.7**	1480(1)	1490.9	1483(2)	1491.8
Media		**1389.8**		1391.1		1391.2

En cuanto al tiempo de ejecución utilizado, los experimentos en los que se evalúa de forma exacta el makespan de todos los vecinos cuya estimación es menor que el makespan del individuo original han consumido alrededor del doble de tiempo, mientras que los experimentos en los que no se utilizan estimaciones han consumido un tiempo aproximadamente catorce veces superior, aunque debemos destacar que en estos dos últimos casos no hemos utilizado algoritmos muy optimizados y por eso la diferencia de tiempos es tan elevada. De

todas formas, podemos ver que a pesar del aumento de tiempo los resultados obtenidos no son mejores, por lo que concluímos que la mejor opción es confiar plenamente en el algoritmo de estimación. Sin embargo, en la sección 10.3.1 veremos que no ocurre lo mismo con las funciones objetivo de tipo suma.

8.3.10. Influencia del operador de mutación

En la sección 5.9 explicamos que no utilizaríamos un operador de mutación en esta tesis. Como esta decisión puede ser bastante discutible dado que hay una gran cantidad de algoritmos genéticos en la literatura que utilizan operadores de mutación, a continuación comprobaremos si dicho operador tiene realmente una influencia importante en el proceso evolutivo.

Al elegir el operador de mutación para realizar estos experimentos, debemos tener en cuenta que estamos utilizando una representación mediante permutaciones con repetición. Esto implica que en muchos operadores de mutación tradicionales, como pueden ser el intercambio de dos posiciones del cromosoma ya sean aleatorias o consecutivas, puede darse que frecuentemente la planificación del individuo mutado sea exactamente igual que la del individuo original.

Por este motivo, optamos por utilizar un operador de mutación que consiste en lo siguiente: elegimos dos puntos aleatorios dentro del cromosoma, y reordenamos aleatoriamente el vector de tareas que se encuentre entre dichos puntos. Este operador provoca una mutación por lo general bastante fuerte, y por lo tanto la probabilidad de que el individuo mutado sea diferente del original es mucho más alta que con los operadores de mutación basados en un único intercambio.

La tabla 8.14 muestra los resultados de estos experimentos. Hemos utilizado los mismos valores de probabilidad de aplicar la mutación propuestos en la experimentación similar realizada por Essafi et al. en [49], es decir, 0.0, 0.05, 0.1 y 0.5. Por otra parte, los parámetros utilizados han sido los mismos de la sección anterior: /60/85/200/ para las instancias LA27sdst y LA29sdst, /80/110/200/ para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst, y /40/60/200/ para todas las restantes instancias.

A partir de estos resultados podemos concluir que cuando utilizamos una combinación de algoritmo genético y búsqueda tabú, el hecho de utilizar un operador de mutación no parece mostrar muchos beneficios. Aunque su influencia en un algoritmo genético simple es posible que sea más relevante, vemos que cuando se combina con una búsqueda local avanzada ya no

Tabla 8.14: Experimentos makespan: Operador de mutación en el algoritmo genético híbrido

	Mutación 0.0		*Mutación 0.05*		*Mutación 0.1*		*Mutación 0.5*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(10)	1440.8	1438(16)	**1440.5**	1438(14)	1440.9	1438(11)	1441.2
t2-ps12	1269(10)	1277.1	1269(3)	1279.4	1269(5)	**1275.1**	1269(1)	1281.3
t2-ps13	1415(28)	1416.0	1415(29)	**1415.3**	1415(26)	1417.0	1415(26)	1416.3
t2-ps14	1452(1)	1488.8	1479(2)	1491.1	1468(1)	1490.7	1473(1)	**1488.6**
t2-ps15	1485(10)	1495.9	1485(9)	**1494.1**	1485(2)	1498.0	1485(4)	1495.5
ABZ7sdst	1231(1)	1245.3	1232(1)	1245.9	1228(2)	**1243.4**	1229(1)	1246.5
ABZ8sdst	1257(1)	**1271.2**	1259(1)	1273.3	1260(1)	1273.8	1263(1)	1277.5
ABZ9sdst	1217(2)	1234.6	1217(1)	1233.2	1217(1)	**1232.7**	1219(1)	1235.9
LA21sdst	1268(6)	1276.0	1268(3)	**1275.9**	1268(7)	1277.0	1268(4)	1278.8
LA24sdst	1151(15)	1152.6	1151(17)	1152.4	1151(15)	1152.9	1151(16)	**1152.3**
LA25sdst	1183(5)	**1187.6**	1183(2)	1187.9	1183(3)	1187.9	1183(4)	**1187.6**
LA27sdst	1735(1)	**1753.7**	1732(1)	1754.0	1732(1)	1756.2	1732(1)	1758.2
LA29sdst	1659(4)	**1666.7**	1651(1)	1667.4	1659(3)	1666.9	1664(20)	1666.8
LA38sdst	1446(17)	1451.7	1446(14)	1452.5	1446(19)	**1450.4**	1446(11)	1453.9
LA40sdst	1482(1)	**1489.7**	1482(1)	1490.7	1483(1)	1491.3	1483(3)	1490.8
Media		**1389.8**		1390.2		1390.3		1391.4

lo es tanto. Como ya hemos comentado, esta misma conclusión se puede observar también en otras publicaciones del estado del arte, por ejemplo en [49].

8.3.11. Influencia de la comprobación de factibilidad

En la sección 7.4.2 hemos definido un algoritmo que puede calcular de forma exacta si un determinado vecino es factible o no. Sin embargo esta comprobación es más compleja y costosa computacionalmente que la condición suficiente definida en la sección 7.4.2. En esta sección se analizará empíricamente si la forma de comprobar la factibilidad tiene una influencia importante en el proceso evolutivo. Los experimentos realizados utilizan de nuevo el mismo híbrido de algoritmo genético y búsqueda tabú de secciones anteriores. La tabla 8.15 muestra el mejor resultado y el resultado medio de tres experimentos: el primero de ellos con la comprobación de factibilidad simple ya utilizada en anteriores secciones, el tercero de ellos utilizando una comprobación de factibilidad exacta, y el segundo un termino medio alcanzado utilizando en el algoritmo el parámetro $maxNodos = 5$ (ver sección 7.4.2). En los tres experimentos utilizamos la misma configuración de población, generaciones e iteraciones de búsqueda tabú de anteriores secciones, sin embargo debemos tener en cuenta

que los experimentos con comprobación exacta de factibilidad han consumido un 39 % más de tiempo de ejecución que la comprobación simple, mientras que los experimentos con $maxNodos = 5$ han consumido un 26 % más de tiempo que la comprobación simple. Sobre la diferencia de tiempos de ejecución, hay que tener en cuenta que no sólo el algoritmo de estimación es más costoso de evaluar, sino que después tenemos más vecinos que tratar y debemos estimar el makespan de todos ellos.

Tabla 8.15: Experimentos makespan: Precisión con la que se comprueba la factibilidad de los vecinos

	Fact. simple		*Fact. media*		*Fact. exacta*	
Instancia	Mejor	Media	Mejor	Media	Mejor	Media
t2-ps11	1438(10)	**1440.8**	1438(10)	1441.8	1438(11)	**1440.8**
t2-ps12	1269(10)	1277.1	1269(5)	**1274.9**	1269(5)	1277.1
t2-ps13	1415(28)	1416.0	1415(28)	**1415.6**	1415(25)	1417.0
t2-ps14	1452(1)	1488.8	1452(2)	**1488.0**	1466(1)	1489.7
t2-ps15	1485(10)	1495.9	1485(4)	1496.3	1485(2)	**1494.1**
ABZ7sdst	1231(1)	1245.3	1228(2)	**1242.6**	1228(1)	1242.8
ABZ8sdst	1257(1)	**1271.2**	1256(1)	1273.0	1261(1)	1274.6
ABZ9sdst	1217(2)	1234.6	1217(3)	**1228.7**	1219(1)	1233.0
LA21sdst	1268(6)	**1276.0**	1268(3)	1277.9	1268(2)	1280.0
LA24sdst	1151(15)	1152.6	1151(18)	**1152.5**	1151(14)	1152.8
LA25sdst	1183(5)	1187.6	1183(5)	**1187.1**	1183(5)	1187.4
LA27sdst	1735(1)	1753.7	1732(2)	**1753.5**	1736(1)	1756.7
LA29sdst	1659(4)	**1666.7**	1651(1)	1668.4	1652(1)	1668.0
LA38sdst	1446(17)	1451.7	1446(22)	**1449.3**	1446(16)	1451.7
LA40sdst	1482(1)	**1489.7**	1469(1)	1490.4	1482(2)	1490.9
Media		1389.8		**1389.3**		1390.4

Como se puede observar en la tabla, los resultados son muy similares en los tres casos, y pensamos que el aumento de tiempo de ejecución no compensa la escasísima diferencia en los resultados. Pensamos que el motivo de que los resultados no mejoren significativamente al comprobar la factibilidad de forma más precisa es que los vecinos que la condición simple descarta como no factibles, pero que en realidad sí que son factibles, son vecinos poco prometedores de todas formas y que no aportarán mucho al resultado final. Por estos motivos seguimos proponiendo el uso de la condición suficiente de factibilidad. Sin embargo, también hemos observado que en algunas instancias muy concretas es beneficiosa una comprobación de factibilidad exacta para los vecinos. Comentaremos un ejemplo de esto en la sección 9.4,

sobre la minimización del maximum lateness.

8.3.12. Influencia de la ordenación topológica

En la sección 5.12 explicamos por qué la forma de ordenar topológicamente la planificación resultante de la búsqueda local podría tener influencia en el proceso evolutivo. En dicha sección hemos comprobado mediante un ejemplo cómo el operador de cruce del algoritmo genético es capaz de generar cuatro hijos diferentes entre sí a partir de dos parejas de padres que representan la misma planificación, sin más que alterar el vector que define su orden topológico. Por este motivo se puede pensar que calcular el orden topológico de forma aleatoria, es decir eligiendo en cada posición del vector una de las tareas disponibles aleatoriamente, puede ser mejor que calcularlo de forma determinista, para aumentar la variedad genética en la población. Para comprobar qué opción es mejor realizamos los experimentos cuyos resultados se muestran en la tabla 8.16. Dichos resultados muestran que las diferencias entre los dos métodos son pequeñas y poco relevantes. Realizar la ordenación topológica de forma aleatoria parece empeorar muy ligeramente las medias, además de hacer un poco más complicado al algoritmo, luego descartamos esa opción.

Tabla 8.16: Experimentos makespan: Distintas formas de realizar la ordenación topológica

	Orden fijo		*Orden aleatorio*	
Instancia	Mejor	Media	Mejor	Media
t2-ps11	1438(10)	**1440.8**	1438(10)	1441.7
t2-ps12	1269(10)	1277.1	1269(6)	**1276.6**
t2-ps13	1415(28)	**1416.0**	1415(27)	1416.3
t2-ps14	1452(1)	**1488.8**	1479(4)	1490.3
t2-ps15	1485(10)	**1495.9**	1485(5)	1496.2
ABZ7sdst	1231(1)	**1245.3**	1240(2)	1252.5
ABZ8sdst	1257(1)	**1271.2**	1268(1)	1280.6
ABZ9sdst	1217(2)	**1234.6**	1220(2)	1241.8
LA21sdst	1268(6)	1276.0	1268(5)	**1275.4**
LA24sdst	1151(15)	**1152.6**	1151(16)	1153.0
LA25sdst	1183(5)	**1187.6**	1183(4)	1187.7
LA27sdst	1735(1)	**1753.7**	1732(1)	1757.9
LA29sdst	1659(4)	1666.7	1651(1)	**1666.5**
LA38sdst	1446(17)	**1451.7**	1446(3)	1458.1
LA40sdst	1482(1)	**1489.7**	1487(7)	1491.6
Media		**1389.8**		1392.4

8.3.13. Influencia del conjunto de caminos críticos a explorar

En la sección 7.5.1 discutimos la posibilidad de utilizar los vecinos de un único camino crítico o los de todos los caminos críticos. Para comparar estas opciones hemos realizado los experimentos cuyos resultados se muestran en la tabla 8.17. La configuración para los experimentos es la misma que la utilizada en anteriores secciones. El hecho de utilizar un único camino crítico únicamente ha conseguido ahorrar entre el 0 % y el 5 % del tiempo de ejecución dependiendo de la instancia. Los resultados son muy similares y no se puede concluir que un método sea mejor que otro. De todas formas que los resultados sean similares en funciones objetivo de tipo bottleneck tiene mucho sentido, porque como ya indicamos en la sección 7.5.1, en una planificación normalmente sólo existe un único camino crítico, y cuando existen varios caminos críticos éstos suelen compartir la mayoría de las tareas. Sin embargo los pocos vecinos extra que tendremos de vez en cuando pueden mejorar el rendimiento de la búsqueda tabú, aunque sea de forma muy ligera.

Tabla 8.17: Experimentos makespan: Utilización de distinto número de caminos críticos

	Todos los caminos		*Un único camino*	
Instancia	Mejor	Media	Mejor	Media
t2-ps11	1438(10)	1440.8	1438(15)	**1439.9**
t2-ps12	1269(10)	1277.1	1269(5)	**1275.3**
t2-ps13	1415(28)	**1416.0**	1415(27)	1416.5
t2-ps14	1452(1)	**1488.8**	1492(30)	1492.0
t2-ps15	1485(10)	1495.9	1485(8)	**1491.7**
ABZ7sdst	1231(1)	**1245.3**	1227(1)	**1245.3**
ABZ8sdst	1257(1)	**1271.2**	1254(2)	1271.4
ABZ9sdst	1217(2)	**1234.6**	1217(3)	1236.9
LA21sdst	1268(6)	**1276.0**	1268(7)	1277.3
LA24sdst	1151(15)	**1152.6**	1151(11)	1153.1
LA25sdst	1183(5)	**1187.6**	1183(4)	1187.7
LA27sdst	1735(1)	**1753.7**	1738(2)	1757.3
LA29sdst	1659(4)	1666.7	1659(5)	**1666.1**
LA38sdst	1446(17)	**1451.7**	1446(14)	1452.3
LA40sdst	1482(1)	**1489.7**	1482(3)	1490.8
Media		**1389.8**		1390.2

8.3.14. Resumen

En esta sección hemos evaluado las diferentes opciones de configuración de parámetros para nuestros métodos. En primer lugar hemos elegido unos valores de tamaño de población y número de generaciones que aseguran la convergencia del algoritmo elegido inicialmente. Hemos visto que la estructura de vecindad que mejores resultados ofrece, a igualdad de tiempos de ejecución, es N^S. Posteriormente comparamos los algoritmos de decodificación y, dependiendo del tipo de ejecución que vayamos a realizar puede ser mejor uno u otro, pero de todas formas si queremos utilizar un planificador activo ofrece mejores resultados uno dominante, luego descartamos el algoritmo $EGYT1$. A continuación hemos visto que el algoritmo híbrido es mejor que la utilización de cualquiera de los dos métodos por separado, y también que la escalada simple es una muy buena alternativa a la escalada de máximo gradiente. Sin embargo, la mejor opción de todas ha resultado ser una combinación de algoritmo genético con búsqueda tabú. Respecto a esto realizamos experimentos para ver cuánto peso es conveniente darle a la búsqueda tabú cuando se combina con un algoritmo genético, y hemos mostrado que una combinación equilibrada es la opción que da mejores resultados, excepto en ejecuciones con un tiempo de ejecución muy reducido, en las cuales el mejor método parece ser la búsqueda tabú por separado.

Una vez que concluímos que la mejor configuración es un híbrido de algoritmo genético y búsqueda tabú utilizando la estructura de vecindad N^S, realizamos varias series de experimentos más para comprobar estrategias alternativas en puntos concretos para disponer de un análisis exhaustivo del algoritmo. En primer lugar probamos los dos tipos clásicos de evolución en algoritmos meméticos: Lamarckiana y Baldwiniana, resultando mejor la primera. Posteriormente comprobamos que la mejor opción es estimar el makespan de toda la vecindad y elegir al vecino con una estimación menor, ya que hemos probado otras dos alternativas y obtienen resultados similares en un tiempo de ejecución mucho mayor. En cuanto al añadido de operadores de mutación al algoritmo genético, hemos llegado a la conclusión de que cuando se combina con una búsqueda local avanzada no aportan mucho y se puede prescindir de ellos. Posteriormente hemos realizado experimentos utilizando una comprobación exacta de la factibilidad de los vecinos generados por la estructura de vecindad, con la conclusión de que es más aconsejable utilizar la condición suficiente rápida de evaluar, ya que los resultados son similares y el tiempo de ejecución más reducido. También experimentamos con la ordenación topológica de los cromosomas generados y comprobamos que una ordenación aleatoria no mejora los resultados obtenidos. Por último vimos que uti-

lizar todos los caminos críticos de la planificación alcanza resultados similares que utilizar un único camino crítico, con un aumento en el tiempo de ejecución casi inapreciable.

8.4. Comparación con los mejores métodos de la literatura

En esta sección comprobaremos si la mejor configuración alcanzada en la sección anterior es competitiva con los mejores algoritmos conocidos en la literatura. Realizaremos experimentos sobre las instancias del conjunto BT, el conjunto propuesto por Cheung y Zhou en [40], el propuesto por Vela et al. en [151], y finalmente sobre varios benchmarks del JSP clásico sin tiempos de setup. Nos compararemos, en cada caso, con los mejores resultados del estado del arte existente para cada benchmark.

8.4.1. Experimentos sobre el conjunto de instancias BT

Para evaluar nuestro método en las instancias del conjunto BT hemos considerado el método de ramificación y poda de Artigues y Feillet propuesto en [15] y el método basado en el heurístico shifting bottleneck de Balas et al. propuesto en [19]. Recordemos que las instancias t2-ps01 hasta t2-ps05 son de tamaño 10×5 (instancias pequeñas), las instancias desde t2-ps06 hasta t2-ps10 son de tamaño 15×5 (instancias medianas), y las instancias t2-ps11 hasta t2-ps15 son de tamaño 20×5 (instancias grandes). En este estudio experimental también hemos utilizado las instancias t2-pss12 y t2-pss13, que son una variante de las instancias t2-ps12 y t2-ps13, de las que sólo difieren en los tiempos de setup.

La tabla 8.18 muestra los resultados de los experimentos. La primera columna de la tabla indica el nombre de la instancia. La segunda y tercera columnas contienen, respectivamente, las mejores cotas inferiores y las mejores cotas superiores conocidas hasta el momento, tal y como se detallan en [15]. Las dos siguientes columnas de la tabla (BSV05) muestran los resultados obtenidos por el algoritmo shifting bottleneck propuesto por Balas et al. en [19]. En las dos siguientes columnas (AF08) incluímos los resultados del método de ramificación y poda de [15]. Las cuatro últimas columnas presentan los resultados de nuestro algoritmo AG+TS: la mejor solución obtenida en 30 ejecuciones, la solución media, la desviación estándar y el tiempo medio de una única ejecución. Hemos utilizado la mejor configuración encontrada en la sección anterior, y los parámetros de ejecución son de /10/12/200/ para las instancias

t2-ps01 a t2-ps05, /20/20/200/ para las instancias t2-ps06 a t2-ps10, y /40/60/200/ para las restantes instancias.

El algoritmo de ramificación y poda propuesto en [15] puede utilizar tres estrategias de resolución diferentes. Los resultados indicados en la tabla 8.18 se refieren al tiempo de CPU y cotas superiores obtenidos utilizando lo que denominan estrategia 1, que es su estrategia básica. Es importante aclarar que el tiempo de ejecución de su algoritmo se limita mediante varios parámetros, como el parámetro NBB que limita la cantidad de nodos visitados. En los experimentos resumidos en la tabla 8.18, este parámetro esta fijado en 500000. En [15] también presentan mejores cotas superiores para las instancias t2-ps08 y t2-ps10 de 966 y 1018, respectivamente. Estas son las mejores cotas superiores conocidas para esas instancias y fueron obtenidas con las estrategias que denominan 2 y 3 dejando al algoritmo de ramificación y poda explorar un máximo de 6 millones de nodos. Dichas estrategias comienzan desde las soluciones obtenidas en la estrategia 1.

Es importante tener en cuenta las diferencias en las máquinas, ya que el algoritmo shifting bottleneck de [19] se implementa en lenguaje C y se ejecuta en un Sun Ultra 60 con procesador UltraSPARC-II a 360MHz, el algoritmo de ramificación y poda de [15] se implementa en C++ y se ejecuta en un PC (no se dan más detalles en ese trabajo), mientras que nuestro algoritmo se ejecuta en un Intel Core 2 Duo a 2.66GHz con 2Gb de RAM, compilador gcc 3.4.4 y Windows XP. Debido a que las máquinas en las que se ejecutan los algoritmos son todas diferentes (procesador, compilador y sistema operativo), la comparación no puede ser del todo precisa.

Sin embargo, hemos ejecutado la implementación del método shifting bottleneck de [19] que se puede encontrar en *http://www.andrew.cmu.edu/neils/tsp* (codificado en C), y los tiempos de ejecución obtenidos en nuestra máquina han sido los siguientes: una media de 65 segundos para las instancias t2-ps01 hasta t2-ps05, una media de 121 segundos para las instancias t2-ps06 hasta t2-ps10, y una media de 280 segundos para las siete instancias restantes. Observamos que los tiempos de ejecución que utiliza nuestro algoritmo son considerablemente más reducidos.

Si comparamos los resultados medios de AG+TS con los de AF08, nuestro algoritmo alcanza un makespan mejor en 7 instancias, el mismo en otras 4, y un makespan peor en las restantes 4. En concreto, hay que aclarar que los mejores valores medios se consiguen principalmente en las instancias más grandes, ya que en cuatro de esas cinco instancias nuestro makespan medio es mejor que el mejor valor alcanzado por AF08. Por otra parte,

Tabla 8.18: Experimentos makespan: Comparación con el estado del arte. Instancias BT

			$BSV05$		$AF08$		$AG+TS$			
Instancia	LB	UB	Mejor	T(s)	Mejor	T(s)	Mejor	Media	Desv	T(s)
t2-ps01	798	798(*)	798(*)	384	798(*)	54	798(30)(*)	798.0(*)	0.0	0.6
t2-ps02	784	784(*)	784(*)	588	784(*)	57	784(30)(*)	784.0(*)	0.0	0.6
t2-ps03	749	749(*)	749(*)	882	749(*)	77	749(30)(*)	749.0(*)	0.0	0.7
t2-ps04	730	730(*)	730(*)	554	730(*)	11	730(28)(*)	730.2	0.8	0.6
t2-ps05	691	691(*)	693	264	691(*)	14	691(18)(*)	691.8	1.0	0.6
t2-ps06	1009	1009(*)	1018	1211	1009(*)	6151	1026(30)	1026.0	0.0	3.8
t2-ps07	970	970(*)	1003	1374	970(*)	10008	970(30)(*)	970.0(*)	0.0	3.6
t2-ps08	958	966	975	846	982	1471	**963(8)**	**965.7**	1.7	3.7
t2-ps09	1051	1060	1060	430	1061	595	1060(30)	1060.0	0.0	4.0
t2-ps10	1018	1018(*)	1018(*)	542	1047	5692	1018(30)(*)	1018.0(*)	0.0	4.0
t2-ps11	1382	1470	1470	3033	1494	8452	**1438(10)**	**1440.8**	2.9	34.9
t2-ps12	1226	1305	1305	2186	1381	1748	**1269(10)**	**1277.1**	14.5	34.2
t2-ps13	1320	1439	1439	2506	1457	12401	**1415(28)**	**1416.0**	3.8	34.2
t2-ps14	1431	1483	1485	2115	1483	3299	**1452(1)**	1488.8	9.0	38.5
t2-ps15	1390	1527	1527	3029	1661	25152	**1486(10)**	**1495.9**	11.6	35.3
t2-pss12	-	1290	1290	2079	-	-	**1258(1)**	**1265.8**	3.8	34.8
t2-pss13	-	1398	1398	1864	-	-	**1361(1)**	**1378.5**	6.6	36.2

Los valores en **negrita** mejoran la mejor solución conocida, (*) solución óptima.

si comparamos nuestros resultados con los de BSV05, nuestro algoritmo alcanza un mejor makespan en 7 instancias, el mismo en otras 5, y valores peores en las restantes 3. Al igual que ocurría en la comparación contra AF08, en cuatro de las cinco instancias grandes AG+TS obtiene un mejor makespan medio que el mejor makespan alcanzado por BSV05. Los valores del error estándar confirman la fiabilidad de estos resultados, con una fluctuación baja de los valores medios obtenidos por AG+TS.

En cuanto a los mejores resultados alcanzados por nuestro algoritmo, observamos que alcanza la solución óptima en la mayoría de las 30 ejecuciones para todas las instancias pequeñas. Para las instancias medias, alcanza la solución óptima en dos casos, la mejor solución conocida en un caso y ha mejorado la mejor solución conocida para la instancia t2-ps08. Para la instancia t2-ps06 no ha sido capaz de alcanzar la mejor solución conocida de 1009 que además es óptima; en las 30 ejecuciones la solución alcanzada fue 1026, como en muchos de los métodos anteriores del estado del arte comentados en [15]. En cuanto a las instancias más difíciles, desde t2-ps11 hasta t2-ps15, AG+TS ha mejorado las mejores soluciones conocidas en todos los casos.

Por último, sobre las instancias t2-pss12 y t2-pss13, aunque en [15] no se ofrecen resultados para ellas, en la tabla se puede ver que nuestro método mejora claramente a BSV05, obteniendo un makespan medio más bajo que el mejor makespan alcanzado por ellos.

8.4.2. Experimentos sobre el conjunto de instancias de Cheung y Zhou

A continuación experimentamos sobre el conjunto de instancias propuesto por Cheung y Zhou en [40]. Los detalles de estas instancias se pueden consultar en la sección 8.2.2. Recordemos que la desigualdad triangular de tiempos de setup no se cumple para estas instancias, por lo que si queremos utilizar el constructor de planificaciones activas SSGS es necesario un mecanismo de reparación (ver sección 3.3.2).

El algoritmo propuesto por Cheung y Zhou en [40] se codifica en FORTRAN y se ejecuta en un PC 486/66. El tiempo de computación utilizado por ellos en los problemas de tamaños 10×10, 20×10 y 20×20 se sitúa alrededor de 16, 30 y 70 minutos por ejecución, respectivamente. Cada ejecución de su algoritmo se finalizó tras 2000 generaciones y realizaron 10 pruebas por cada instancia. En la tabla 8.19 presentamos los resultados de su algoritmo denominado GA-SPTS. En el mismo artículo proponen otro denominado GA-MWKR pero los resultados experimentales son ligeramente peores, por lo que no los incluímos aquí. En la misma tabla también presentamos los resultados de nuestro híbrido de algoritmo genético con búsqueda tabú con la estructura de vecindad N^S. La configuración elegida es /100/100/200/, y de nuevo no gestionamos la lista de soluciones elite debido a que el número de iteraciones de búsqueda tabú es reducido. Realizamos 30 ejecuciones para cada instancia y presentamos como es habitual el mejor makespan obtenido, la media de makespan, la desviación estándar y el tiempo medio en segundos de una única ejecución. Aunque los tiempos de ejecución no se pueden comparar directamente debido a la gran diferencia entre las máquinas, observamos que en todas las 45 instancias el makespan medio obtenido por nuestro método es más bajo que el obtenido por GA-SPTS. De hecho, el makespan medio obtenido por nuestro método es más bajo que el mejor makespan obtenido por GA-SPTS entre las 10 pruebas que realizan para cada instancia. Además, la diferencia es más amplia cuanto mayor es el tamaño de la instancia. Las desviaciones estándar de nuestro método también son mucho más reducidas, lo que indica que los resultados además son más estables.

Tabla 8.19: Experimentos makespan: Instancias de Cheung y Zhou

			GA-SPTS			*AG+TS /100/100/200/*			
Instancia	Tamaño	Tipo	Mejor	Media	Desv	Mejor	Media	Desv	T(s)
ZRD01	10×10	1	858	883	18.6	814(18)	**816.0**	2.5	56
ZRD02	10×10	1	883	908	15.4	792(8)	**792.7**	0.5	65
ZRD03	10×10	1	847	894	34.3	787(30)	**787.0**	0.0	58
ZRD04	10×10	1	799	814	15.3	747(30)	**747.0**	0.0	60
ZRD05	10×10	1	790	822	23.7	736(4)	**738.5**	1.1	58
ZRD06	10×10	2	1334	1334	1.8	1241(13)	**1243.3**	2.1	55
ZRD07	10×10	2	1327	1395	47.8	1247(6)	**1250.3**	2.5	57
ZRD08	10×10	2	1299	1311	11.8	1209(1)	**1215.7**	1.4	52
ZRD09	10×10	2	1326	1365	22.9	1269(15)	**1271.0**	2.0	56
ZRD10	10×10	2	1329	1343	20.7	1284(2)	**1288.1**	1.5	55
ZRD11	10×10	3	1538	1566	45.3	1448(30)	**1448.0**	0.0	54
ZRD12	10×10	3	1609	1650	39.1	1497(30)	**1497.0**	0.0	59
ZRD13	10×10	3	1442	1484	51.4	1366(30)	**1366.0**	0.0	51
ZRD14	10×10	3	1519	1519	0.0	1434(30)	**1434.0**	0.0	58
ZRD15	10×10	3	1515	1561	41.4	1424(30)	**1424.0**	0.0	55
ZRD16	20×10	1	1316	1338	17.1	1146(11)	**1150.3**	3.9	146
ZRD17	20×10	1	1337	1414	38.5	1146(2)	**1151.3**	3.6	145
ZRD18	20×10	1	1350	1383	19.2	1126(2)	**1131.9**	2.5	144
ZRD19	20×10	1	1353	1385	39.9	1143(1)	**1157.0**	9.0	141
ZRD20	20×10	1	1341	1365	11.9	1115(1)	**1119.8**	2.9	144
ZRD21	20×10	2	2359	2429	67.8	2073(1)	**2091.7**	6.1	140
ZRD22	20×10	2	2345	2400	53.5	2021(1)	**2030.2**	4.5	135
ZRD23	20×10	2	2389	2422	43.3	2047(3)	**2057.0**	6.4	140
ZRD24	20×10	2	2217	2254	26.1	1992(1)	**1998.6**	5.0	133
ZRD25	20×10	2	2316	2374	40.6	2062(4)	**2066.3**	2.3	135
ZRD26	20×10	3	2591	2641	34.0	2225(2)	**2235.1**	7.5	150
ZRD27	20×10	3	2442	2530	81.1	2104(7)	**2108.7**	3.4	138
ZRD28	20×10	3	2363	2410	28.7	2094(2)	**2109.5**	5.9	160
ZRD29	20×10	3	2366	2514	108.8	2091(7)	**2096.2**	3.6	136
ZRD30	20×10	3	2371	2536	126.9	2104(2)	**2110.5**	3.8	143
ZRD31	20×20	1	1773	1835	52.7	1498(1)	**1517.1**	6.7	198
ZRD32	20×20	1	1951	2011	43.7	1457(3)	**1466.8**	7.2	207
ZRD33	20×20	1	1815	1865	46.3	1487(1)	**1504.5**	7.1	193
ZRD34	20×20	1	1673	1723	57.6	1439(3)	**1448.4**	7.5	190
ZRD35	20×20	1	1725	1813	88.6	1457(1)	**1469.1**	7.0	201
ZRD36	20×20	2	3007	3155	95.3	2438(1)	**2451.6**	8.4	193
ZRD37	20×20	2	3004	3098	101.9	2535(1)	**2570.1**	18.6	197
ZRD38	20×20	2	2706	2826	109.9	2312(1)	**2346.9**	14.5	185
ZRD39	20×20	2	2698	2786	56.9	2355(2)	**2372.6**	10.7	189
ZRD40	20×20	2	2882	3045	105.6	2480(2)	**2506.1**	16.2	196
ZRD41	20×20	3	3117	3206	113.2	2677(1)	**2685.7**	5.6	182
ZRD42	20×20	3	3202	3317	78.5	2773(2)	**2789.5**	9.1	195
ZRD43	20×20	3	3248	3296	44.0	2801(1)	**2824.2**	11.2	193
ZRD44	20×20	3	3140	3382	298.4	2752(1)	**2774.5**	10.6	203
ZRD45	20×20	3	3282	3347	73.5	2820(1)	**2833.4**	7.6	189

8.4.3. Experimentos sobre el conjunto de instancias de Vela et al.

En esta sección consideramos el benchmark propuesto por Vela et al. en [151], descrito en la sección 8.2.3. Debido a que es un benchmark que en la literatura no ha sido utilizado por otros grupos de investigación, hemos decidido modelar el problema mediante el ILOG CPLEX CP Optimizer (CP) y comparar nuestros resultados con los obtenidos por dicho método. Los detalles del método CP se pueden consultar en la sección 4.4.3.

La configuración utilizada para nuestro algoritmo $AG + TS$ depende del tamaño de la instancia y es el siguiente (/ tamaño de la población / número de generaciones / iteraciones de búsqueda tabú /):

- /60/85/200/ para las instancias LA27sdst y LA29sdst.
- /80/110/200/ para las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst.
- /40/60/200/ para todas las restantes instancias.

Por otra parte, en los experimentos hemos dejado ejecutarse CP aproximadamente el doble de tiempo que el utilizado por $AG + TS$. La tabla 8.20 muestra los resultados de esta serie de experimentos. En concreto indicamos, para cada uno de los dos métodos, el mejor makespan alcanzado en las 30 ejecuciones y el número de veces que se alcanza, el makespan medio y la desviación estándar, además del tiempo medio en segundos de una única ejecución. Tanto con CP como con $AG + TS$ hemos realizado 30 ejecuciones para cada una de las instancias.

Tabla 8.20: Experimentos makespan: Resultados en el benchmark propuesto por Vela et al.

Inst.	Tamaño	CP				$AG + TS$			
		Mejor	Media	Desv	T(s)	Mejor	Media	Desv	T(s)
ABZ7sdst	20×15	1253(1)	1284.6	14.0	350	1231(1)	**1245.3**	7.6	180
ABZ8sdst	20×15	1279(1)	1311.2	14.4	350	1257(1)	**1271.2**	8.0	175
ABZ9sdst	20×15	1259(1)	1300.0	19.3	350	1217(2)	**1234.6**	12.7	176
LA21sdst	15×10	1299(1)	1333.5	13.1	50	1268(6)	**1276.0**	7.0	26
LA24sdst	15×10	1156(1)	1186.6	16.9	50	1151(15)	**1152.6**	1.7	23
LA25sdst	15×10	1193(1)	1222.2	16.9	50	1183(5)	**1187.6**	2.7	26
LA27sdst	20×10	1763(1)	1807.1	20.2	150	1735(1)	**1753.7**	10.8	73
LA29sdst	20×10	1686(1)	1717.8	24.2	150	1659(4)	**1666.7**	6.1	78
LA38sdst	15×15	1491(1)	1540.8	23.2	65	1446(17)	**1451.7**	7.7	33
LA40sdst	15×15	1496(1)	1523.9	14.4	65	1482(1)	**1489.7**	4.3	33

Observamos que tanto el mejor makespan como el makespan medio obtenidos por $AG + TS$ son mejores que los obtenidos por CP, a pesar de que el tiempo de ejecución medio es más reducido. De hecho, en todos los casos la media de makespan obtenida por $AG + TS$ es más baja que el mejor makespan alcanzado por CP en sus 30 ejecuciones. Podemos ver también que las desviaciones estándar son en general más reducidas, lo que indica que $AG + TS$, además de obtener mejores valores de makespan, es un método más estable y robusto ya que los consigue con una menor variabilidad.

8.4.4. Experimentos sobre instancias del job-shop clásico

Para completar el estudio experimental sobre el makespan hemos lanzado ejecuciones sobre conjuntos de instancias del job-shop clásico sin tiempos de setup. Utilizaremos primero el conjunto de 10 instancias elegidas por Applegate y Cook en [9] como difíciles de resolver para el JSP, además de la instancia clásica FT10. Para estos experimentos utilizamos de nuevo una combinación de algoritmo genético y búsqueda tabú, con tres diferentes combinaciones de parámetros (/ tamaño de la población de AG / número de generaciones de AG / *maxGlobalIter* de TS /): /40/60/100/, /60/60/200/ y /100/100/200/. Realizamos tres series de experimentos para comprobar la mejora de los resultados en relación al incremento en tiempo de ejecución.

En la tabla 8.21 mostramos los resultados de estos experimentos, junto con los resultados obtenidos en estas instancias por el método de búsqueda tabú combinado con enfriamiento simulado propuesto por Zhang et al. en [162]. Elegimos este método en particular por ser bastante reciente, y por ser uno de los que mejores resultados obtiene en la literatura. En las dos primeras columnas mostramos el nombre de la instancia y su solución óptima. En el caso de las instancias ABZ8 y ABZ9 no se conoce la solución óptima, y entonces indicamos la mejor cota superior conocida, y entre paréntesis la mejor cota inferior conocida. El resto de columnas muestran los resultados de los experimentos con el algoritmo propuesto por Zhang et al. y con nuestro algoritmo en el formato habitual, es decir mejor solución, solución media y tiempo consumido por cada una de las 30 pruebas ejecutadas. En el caso del algoritmo de Zhang et al. debemos notar que sólo realizan 10 pruebas por cada instancia, y que el tiempo de ejecución que indican en su trabajo es comparable al nuestro, ya que utilizan para sus experimentos un Pentium 4 con un procesador de 3.0 GHz., que es una máquina con una potencia comparable a la nuestra (recordemos que nuestro algoritmo se ejecuta en un Intel Core 2 Duo a 2.66GHz).

Tabla 8.21: Experimentos makespan: Comparación con el estado del arte en 11 instancias difíciles del job-shop clásico

					AG+TS			*AG+TS*			*AG+TS*		
		Zhang et al.			*/40/60/100/*			*/60/60/200/*			*/100/100/200/*		
Inst.	UB(LB)	Mejor	Media	T	Mejor	Media	T	Mejor	Media	T	Mejor	Media	T
ABZ7	656	658	661.8	86	666	669.0	27	658	665.8	74	658	662.7	199
ABZ8	665(645)	667	670.3	91	671	679.2	28	669	674.1	76	668	670.8	204
ABZ9	678(661)	678	684.8	90	685	690.4	26	678	685.1	71	678	682.7	194
FT10	930	930	930.0	4	930	930.7	10	930	930.3	27	930	930.0	74
LA21	1046	1046	1046.0	15	1046	1048.4	13	1046	1046.3	38	1046	1046.1	106
LA24	935	935	936.2	20	935	939.8	11	935	937.3	33	935	937.4	90
LA25	977	977	977.1	14	977	979.4	13	977	977.2	37	977	977.2	106
LA27	1235	1235	1235.0	12	1235	1243.8	19	1235	1236.6	53	1235	1235.8	146
LA29	1152	1153	1159.2	64	1162	1169.4	19	1158	1165.7	55	1153	1162.7	149
LA38	1196	1196	1199.6	48	1196	1205.4	17	1196	1199.3	47	1196	1197.1	132
LA40	1222	1224	1224.5	52	1224	1227.6	17	1224	1226.6	45	1222	1225.4	127

Los resultados muestran que, como era de esperar, a medida que aumentamos los parámetros de AG+TS los resultados mejoran a la par que el tiempo de ejecución. Utilizando la configuración /40/60/100/ llegamos al óptimo o a la mejor cota superior conocida en 6 de las 11 instancias, mientras que utilizando /60/60/200/ llegamos en 7 instancias, y utilizando /100/100/200/ en 8 instancias.

El método de Zhang et al. consigue el óptimo o a la mejor cota superior conocida en 7 de las 11 instancias. Además, en tiempos de ejecución comparables obtiene por lo general medias de makespan ligeramente inferiores que nuestro algoritmo genético híbrido. Aunque las diferencias son bastante pequeñas, su método ofrece resultados ligeramente mejores en un tiempo ligeramente menor. Sin embargo debemos decir en nuestro favor que el algoritmo genético y la estructura de vecindad utilizada por la búsqueda tabú no están optimizados para tratar con tiempos de setup nulos.

Por último, realizamos un estudio experimental sobre las instancias de Taillard. Éste es un conjunto de 80 instancias con un número de operaciones comprendido entre 225 y 2000. Debido a que la proporción entre el número de trabajos y el número de máquinas va aumentando, las instancias desde la 51 a la 80 son relativamente fáciles de resolver, exceptuando quizás las instancias TA62 y TA67. Por el contrario, las instancias TA01-50 son muy difíciles. De esas 50 instancias sólo se conoce la solución óptima de 18 de ellas. Las

instancias TA21-30 (20×20) y TA41-50 (30×20) son reconocidos como los benchmarks más difíciles para el JSP, y no se ha encontrado ninguna solución óptima para ellas. Al igual que en los experimentos anteriores utilizamos una combinación de algoritmo genético y búsqueda tabú, pero esta vez con los parámetros /100/100/500/ debido a la dificultad de las instancias de este benchmark.

La tabla 8.22 resume los resultados para las instancias difíciles, es decir, TA01-50, TA62 y TA67. En las 3 primeras columnas mostramos el nombre de la instancia, su tamaño y la solución óptima. En el caso de que dicha solución óptima no se conozca, indicamos la mejor cota superior conocida y entre paréntesis la mejor cota inferior conocida. Con el objetivo de mostrar los datos más recientes sobre las mejores cotas, hemos utilizado los presentados por Beck et al. en [26], actualizados con las nuevas mejores cotas superiores que ellos mismos obtienen. Dicho trabajo es del año 2010. Las tres siguientes columnas indican el resultado obtenido por el método de Zhang et al. propuesto en [162], y las tres últimas columnas de la tabla muestran los resultados de nuestro algoritmo. Debemos observar que para las instancias TA62 y TA67, Zhang et al. no ofrecen datos del makespan medio ni del tiempo de ejecución medio obtenidos por ellos.

Observamos que, de las 20 instancias que mostramos para las que se conoce la solución óptima, nuestro algoritmo es capaz de alcanzar dicha solución óptima en 14 de esas 20 instancias. Para las restantes instancias, los resultados de nuestro algoritmo se sitúan siempre bastante cerca de la mejor cota superior conocida, siendo capaz de igualarla en las instancias TA11, TA16, TA22 y TA29.

En cuanto a la comparación con el método propuesto en [162], en las instancias TA01 a TA50 nuestro método ha obtenido una mejor media de makespan en 17 instancias, igual en 1 instancia, y peor en 32 instancias. El tiempo de ejecución consumido por nuestro algoritmo es en general algo mayor, especialmente en las instancias TA01 a TA10. Por todos estos motivos concluímos que el método de Zhang et al. es más eficiente en la minimización del makespan en el JSP clásico, sin embargo nuestro método es competitivo a pesar de no estar optimizado para resolver específicamente dicho problema.

Tabla 8.22: Experimentos makespan: Instancias de Taillard del job-shop clásico

			Zhang et al.			*AG+TS /100/100/500/*		
Instancia	Tamaño	UB(LB)	Mejor	Media	T(s)	Mejor	Media	T(s)
TA01	15×15	1231	1231	1230.0	11	1231(29)	1231.3	213
TA02	15×15	1244	1244	1244.1	30	1244(18)	1244.4	234
TA03	15×15	1218	1218	1219.4	109	1218(25)	1218.6	232
TA04	15×15	1175	1175	1176.2	72	1175(7)	1179.6	237
TA05	15×15	1224	1224	1224.0	11	1224(19)	1226.6	248
TA06	15×15	1238	1238	1240.8	125	1238(24)	1238.2	239
TA07	15×15	1227	1228	1228.0	139	1228(30)	1228.0	223
TA08	15×15	1217	1217	1217.1	28	1217(30)	1217.0	222
TA09	15×15	1274	1274	1275.2	61	1274(27)	1274.6	240
TA10	15×15	1241	1241	1246.6	68	1241(23)	1241.5	237
TA11	20×15	1357(1323)	1359	1367.6	260	1357(1)	1367.0	378
TA12	20×15	1367(1351)	1371	1374.3	253	1371(2)	1376.8	311
TA13	20×15	1342(1282)	1342	1355.2	259	1344(1)	1348.6	321
TA14	20×15	1345	1345	1346.7	65	1345(30)	1345.0	357
TA15	20×15	1339(1304)	1339	1348.4	253	1340(7)	1349.4	345
TA16	20×15	1360(1302)	1360	1366.2	253	1360(30)	1360.0	331
TA17	20×15	1462	1464	1472.9	249	1464(1)	1474.3	366
TA18	20×15	1396(1369)	1399	1408.7	232	1399(2)	1407.0	362
TA19	20×15	1332(1297)	1335	1340.6	262	1337(4)	1341.3	371
TA20	20×15	1348(1318)	1350	1356.7	264	1351(2)	1360.4	337
TA21	20×20	1642(1539)	1644	1650.5	437	1647(14)	1652.7	423
TA22	20×20	1600(1511)	1600	1606.4	434	1600(1)	1613.5	409
TA23	20×20	1557(1472)	1560	1564.5	429	1558(1)	1564.1	395
TA24	20×20	1645(1602)	1646	1653.2	432	1652(1)	1653.9	434
TA25	20×20	1595(1504)	1597	1607.7	421	1597(1)	1607.5	408
TA26	20×20	1645(1539)	1647	1654.9	436	1650(1)	1654.8	444
TA27	20×20	1680(1616)	1680	1688.7	448	1689(5)	1691.7	434
TA28	20×20	1603(1591)	1603	1616.6	431	1616(1)	1617.4	399
TA29	20×20	1625(1514)	1627	1630.1	426	1625(15)	1626.5	397
TA30	20×20	1584(1473)	1584	1597.7	436	1589(2)	1599.4	437
TA31	30×15	1764	1764	1765.8	318	1764(21)	1764.5	601
TA32	30×15	1795(1774)	1795	1811.7	613	1803(1)	1816.5	592
TA33	30×15	1791(1778)	1796	1806.7	495	1806(1)	1818.1	595
TA34	30×15	1829(1828)	1831	1831.8	338	1832(1)	1834.9	595
TA35	30×15	2007	2007	2011.0	130	2007(30)	2007.0	635
TA36	30×15	1819	1819	1820.5	178	1819(7)	1822.1	592
TA37	30×15	1771	1778	1784.4	496	1789(2)	1794.7	578
TA38	30×15	1673	1673	1678.5	333	1676(1)	1687.6	592
TA39	30×15	1795	1795	1806.6	303	1799(2)	1806.1	632
TA40	30×15	1673(1631)	1676	1684.0	499	1688(1)	1696.2	606
TA41	30×20	2012(1859)	2018	2028.5	806	2029(1)	2046.2	734
TA42	30×20	1949(1867)	1953	1964.5	805	1964(2)	1971.9	727
TA43	30×20	1858(1809)	1858	1882.6	923	1874(1)	1887.8	711
TA44	30×20	1983(1927)	1983	1998.2	898	2004(1)	2015.9	749
TA45	30×20	2000(1997)	2000	2006.8	805	2008(1)	2013.6	707
TA46	30×20	2015(1940)	2010	2029.2	819	2027(1)	2045.6	772
TA47	30×20	1903(1789)	1903	1918.2	877	1916(1)	1931.8	723
TA48	30×20	1949(1912)	1955	1968.6	803	1965(1)	1983.0	714
TA49	30×20	1967(1915)	1967	1980.0	911	1981(1)	1994.5	733
TA50	30×20	1926(1807)	1931	1945.6	812	1942(1)	1954.7	703
TA62	50×20	2869	2869	-	-	2879(1)	2897.4	1523
TA67	50×20	2825	2825	-	-	2825(18)	2825.4	1441

8.5. Conclusiones

En este capítulo hemos tratado la minimización del makespan en el SDST-JSP. Para empezar hemos realizado un estudio experimental para elegir la mejor configuración de entre todos los métodos y parámetros presentados a lo largo de esta tesis. La configuración que mejores resultados obtuvo para la minimización del makespan es un algoritmo genético combinado con búsqueda tabú, utilizando la estructura de vecindad N^S. Posteriormente hemos comparado esta combinación con los mejores métodos del estado del arte. En particular, nos hemos comparado en el conjunto de instancias BT con el algoritmo shifting bottleneck propuesto por Balas et al. en [19] y con el algoritmo de ramificación y poda propuesto por Artigues y Feillet en [15]. El resultado de los experimentos demuestra que nuestro método es competitivo con el resto de métodos del estado del arte, logrando mejorar la mejor solución conocida en 6 de las 15 instancias del conjunto. Posteriormente hemos realizado experimentos sobre el benchmark propuesto por Cheung y Zhou en [40], logrando superar los resultados presentados en dicho trabajo. También hemos considerado el benchmark propuesto por Vela et al. en [151], con instancias derivadas de las elegidas por Applegate y Cook en [9] como difíciles de resolver para el JSP, y nuestro método mejora los resultados obtenidos por el ILOG CPLEX CP Optimizer. Por último hemos completado el estudio experimental con resultados sobre conjuntos de instancias del JSP clásico, demostrando que en ausencia de tiempos de setup el algoritmo propuesto también se comporta de forma sobresaliente. Sin embargo, en este último caso no ha sido capaz de mejorar los resultados obtenidos por el método propuesto por Zhang et al. en [162], pero debemos tener en cuenta que nuestro algoritmo genético híbrido no esta optimizado para tratar con tiempos de setup nulos.

Capítulo 9

ESTUDIO EXPERIMENTAL. MINIMIZACIÓN DEL MAXIMUM LATENESS

9.1. Introducción

En este capítulo describiremos el estudio experimental realizado para la minimización del maximum lateness. Para esta función objetivo existe un conjunto de instancias sobre el que se comparan muchos de los autores, por lo que éste que será el benchmark que utilizaremos en este análisis. No realizaremos de nuevo el ajuste de parámetros y, para elegir la mejor configuración para nuestro algoritmo, aprovecharemos el estudio experimental realizado en el capítulo 8, ya que la minimización del makespan y del maximum lateness es prácticamente equivalente, como hemos ido poniendo de manifiesto en el trabajo previo presentado en la tesis. Así pues, el propósito fundamental de este capítulo es comparar nuestros métodos para minimizar el maximum lateness con los mejores métodos del estado del arte que, por lo que sabemos, son el algoritmo shifting bottleneck con búsqueda local guiada (al que denominaremos SB-GLS) propuesto por Balas et al. en [20] y el algoritmo de búsqueda por muestreo iterativo (en adelante ISS, Iterative Sampling Search) propuesto por Oddi et al. en [107].

9.2. Descripción del benchmark utilizado

Utilizaremos los conjuntos de instancias propuestos por Ovacik y Uzsoy en [109], conocidos por ser difíciles de resolver. En todas las instancias de estos benchmarks, los tiempos de setup S_{ij} y la duración p_{ij} de cada tarea son valores aleatorios en el intervalo [1, 200]. El due date d_i de los trabajos se considera uniformemente distribuído en un intervalo I que se caracteriza por los dos siguientes parámetros:

1. el valor medio $\mu = (1 - \tau)E[C_{max}]$, donde τ (tardiness) representa el porcentaje de los trabajos que se espera que no cumplan el due date, y $E[C_{max}]$ es el makespan esperado (calculado estimando el tiempo total de setup y de procesamiento requerido por todos los trabajos en todas las máquinas y dividiendo el resultado por el número de máquinas disponibles).

2. el valor R (spread) que determina la amplitud de I, y cuyos extremos están definidos por: $[\mu - R/2, \mu + R/2]$.

Todas las instancias del benchmark se generan utilizando los valores $\tau = 0{,}3$ y $\tau = 0{,}6$, que corresponden a due dates permisivos y restrictivos, respectivamente, y los valores $R = 0{,}5$, $R = 1{,}5$ y $R = 2{,}5$, que modelan diferentes variaciones en los due dates. La combinación de todos estos valores de τ y de R permite agrupar las instancias en seis benchmarks diferentes, llamados: $i305$, $i315$, $i325$, $i605$, $i615$, $i625$. Cada benchmark contiene 160 instancias generadas aleatoriamente, divididas en subclases según el número de trabajos y de máquinas de la instancia; concretamente, las instancias pueden tener 10 ó 20 trabajos, y 5, 10, 15 ó 20 máquinas, dando lugar a 8 subclases de instancias para cada benchmark. Por último, hay que destacar que estos benchmarks no satisfacen la desigualdad triangular de tiempos de setup, ya que como hemos comentado anteriormente, los tiempos de setup son valores aleatorios en el intervalo [1, 200].

9.3. Descripción del estudio experimental realizado

Para comparar nuestro algoritmo con el método SB-GLS de Balas [20], utilizamos los resultados que se ofrecen en *http://www.andrew.cmu.edu/user/neils/tsp/* cuando el parámetro K se fija a 15. Dicho parámetro se utiliza en su algoritmo de programación dinámica, y elegimos $K = 15$ porque es con el que obtiene la mejor solución en media.

Dado que con el grafo disyuntivo propuesto, la resolución del maximum lateness es equivalente a la del makespan, podemos asumir que la información obtenida a lo largo del estudio experimental sobre la minimización del makespan será aplicable también aquí. SB-GLS utiliza unos tiempos de ejecución muy reducidos en general, y como queremos compararnos en unos tiempos de ejecución similares, la mejor opción será utilizar una búsqueda tabú, ya que hemos visto en la sección 8.3.7 que la combinación de algoritmo genético y búsqueda tabú necesita un tiempo de ejecución mayor para mejorar la calidad de las soluciones de la búsqueda tabú por separado. Entonces, utilizamos el modelo de búsqueda tabú descrito en la sección 6.4, comenzando desde una solución activa aleatoria generada por SSGS (ver sección 3.3.2), y utilizando la lista de soluciones elite con un tamaño de 10 y el resto de parámetros indicados en la sección 6.4. Por otra parte utilizamos la estructura de vecindad N^S y el método de estimación descritos en el capítulo 7. Denominaremos a partir de este momento $TS - N^S$ al algoritmo propuesto para este estudio experimental.

El tiempo de ejecución que le damos a $TS - N^S$ (codificado en C++) es similar al tiempo de ejecución utilizado por SB-GLS [20]. Para establecer dicho tiempo hemos compilado y ejecutado en nuestra máquina (Intel Core 2 Duo a 2.6GHz, compilador gcc 3.4.4 y Windows XP) el código de la implementación de SB-GLS que se puede encontrar en *http://www.andrew.cmu.edu/neils/tsp* (codificado en C), y hemos anotado el tiempo de ejecución utilizado por SB-GLS para cada subfamilia de instancias. Posteriormente elegimos el número máximo de iteraciones de $TS - N^S$ para que el tiempo de ejecución en la misma máquina, y utilizando el mismo compilador, sea menor o igual en todas las instancias. La tabla 9.1 muestra el número máximo de iteraciones elegido para cada uno de los grupos de instancias. Nótese que este valor cambia mucho de unos grupos de instancias a otros porque el método SB-GLS tiene claramente una fuerte dependencia del número de trabajos. Con estos parámetros, de media $TS - N^S$ ha consumido alrededor del 60 % del tiempo utilizado por SB-GLS y en ningún caso el tiempo consumido por $TS - N^S$ ha sido mayor que el de SB-GLS.

El algoritmo propuesto por Oddi et al. en [107] esta implementado en Allegro Common LISP 6.0 y se ejecuta en una máquina a 0.9 Ghz; el tiempo de ejecución varía desde 40 segundos para las instancias más pequeñas hasta un máximo de 3200 segundos para las más grandes. En este caso ya es más difícil establecer tiempos de ejecución equivalentes.

Posteriormente realizaremos una segunda serie de experimentos en la que dejamos a $TS - N^S$ ejecutarse durante un número de iteraciones mucho mayor que en la primera serie

Tabla 9.1: Experimentos maximum lateness: Número de iteraciones de $TS - N^S$ según el grupo de instancias

		Benchmark					
Inst.	Tamaño	i305	i315	i325	i605	i615	i625
1-20	10×5	5000	3500	3500	4000	5000	6000
21-40	20×5	60000	120000	60000	70000	90000	90000
41-60	10×10	12000	10000	5000	12000	12000	10000
61-80	20×10	100000	100000	75000	100000	120000	130000
81-100	10×15	25000	15000	10000	25000	20000	20000
101-120	20×15	75000	65000	50000	80000	80000	75000
121-140	10×20	30000	20000	8000	30000	30000	20000
141-160	20×20	65000	50000	50000	70000	70000	55000

de experimentos, con el objetivo de comprobar su capacidad de mejorar si dispone de un tiempo de ejecución mayor.

En la siguiente sección ofrecemos una tabla resumen con los resultados de la primera serie de experimentos realizada y comentarios generales, y en posteriores secciones realizaremos un análisis más detallado de estos experimentos. En este capítulo realizamos un análisis de los resultados mucho más exhaustivo que en otros capítulos debido al gran tamaño del benchmark utilizado, que permitirá obtener conclusiones dependiendo de los parámetros utilizados para generar cada grupo de instancias.

9.4. Resultados de los experimentos

La tabla 9.2 resume los resultados obtenidos por SB-GLS e ISS, además de los resultados de nuestro algoritmo, $TS - N^S$. La idea es mostrar primero los resultados globales del estudio experimental, y en las siguientes secciones realizar estudios más detallados. En la tabla 9.2, en todos los casos los resultados se promedian para cada grupo de 20 instancias, tal como se indica en la primera columna de la tabla. Los tamaños de las instancias se pueden consultar en la tabla 9.1, presentada en la sección anterior. La segunda columna muestra los resultados obtenidos por el algoritmo SB-GLS propuesto en [20] y la tercera columna muestra los resultados del método ISS propuesto en [107]. Las cuatro últimas columnas muestran los resultados de $TS - N^S$ cuando se le deja ejecutarse un tiempo de ejecución equivalente al utilizado por SB-GLS. En este caso mostramos la mejor solución alcanzada en 30 ejecuciones, la solución media y la desviación estándar de las 30 soluciones, y el tiempo

de ejecución medio de una única ejecución. En la tabla marcamos en negrita las medias de maximum lateness que superan al resto de métodos.

De esta serie de experimentos podemos observar que la mejor solución alcanzada por $TS-N^S$ mejora a las mejores soluciones alcanzadas por SB-GLS e ISS en todos y cada uno de los 48 grupos de instancias. En cuando a los valores medios alcanzados por $TS-N^S$, son mejores que los obtenidos por SB-GLS en 47 grupos del total de 48 grupos. Sólo en el grupo $41-60$ del benchmark i325 el valor alcanzado por SB-GLS es mejor que el valor medio alcanzado por $TS-N^S$. Además, $TS-N^S$ es mejor que ISS en 46 de los 48 grupos. En este caso, ISS es mejor que los valores medios de $TS-N^S$ en los grupos $81-100$ y $121-140$ del benchmark i325. Hay que remarcar que $TS-N^S$ consigue las mayores mejoras con respecto a SB-GLS e ISS para las instancias con el mayor número de trabajos.

También hemos calculado los intervalos de confianza al 95 % para el maximum lateness medio obtenido. La cota superior del intervalo de confianza es menor que los resultados de SB-GLS en 33 de los 48 grupos (68,8 %) y es menor que los resultados de ISS en 36 de los 48 grupos (75 %).

Por otra parte, en algunas de las instancias hemos observado un caso curioso respecto a la utilización de la comprobación exacta de la factibilidad frente a la utilización de la condición suficiente. Por ejemplo, en la instancia i325_42, con la condición suficiente de factibilidad nuestro algoritmo llega al mejor maximum lateness conocido para esa instancia en el 89.5 % de las ejecuciones, mientras que comprobando de forma exacta la factibilidad de los vecinos se ha llegado al mejor maximum lateness conocido en el 99.9 % de las ejecuciones (hemos realizado 10000 ejecuciones con cada configuración para realizar esta comparación). Esto es debido a la estructura de la instancia: hemos comprobado que la ejecución llega frecuentemente a una planificación cuyo camino crítico consta únicamente de un bloque crítico de tamaño 2, y el único vecino posible, aunque es factible, es descartado como no factible por la condición suficiente de factibilidad, y por lo tanto sin ningún vecino posible la búsqueda se queda bloqueada. Sin embargo ese único vecino realmente es factible, y continuando la búsqueda a través de él se acaba llegando fácilmente a la mejor solución conocida. Sin embargo estos casos ocurren muy rara vez y en instancias muy concretas, por lo que seguimos proponiendo la utilización de la condición suficiente de factibilidad.

Tabla 9.2: Experimentos maximum lateness: Resultados de los experimentos

	SB-GLS	ISS	$TS-N^S$			
Inst.			Mejor	Media	Desv	T(s)
i305						
1-20	361.7	372.2	305.4	**313.0**	8.6	0.1
21-40	93.9	294.3	-122.2	**-90.9**	15.0	3.9
41-60	1087.1	1102.3	1010.9	**1032.4**	18.8	0.4
61-80	815.2	1041.0	526.6	**572.6**	22.0	6.6
81-100	1650.7	1670.4	1577.5	**1597.5**	13.0	1.0
101-120	1521.0	1933.8	1205.3	**1281.2**	36.5	6.8
121-140	2158.9	2170.0	2077.0	**2097.9**	14.3	1.5
141-160	2097.0	2908.1	1815.4	**1906.1**	49.1	6.7
i315						
1-20	425.2	425.0	407.6	**414.9**	6.5	0.1
21-40	202.6	185.2	67.5	**85.9**	9.9	3.5
41-60	1081.5	1094.8	1060.0	**1068.3**	11.1	0.3
61-80	757.4	679.9	621.1	**646.8**	20.0	4.3
81-100	1597.5	1589.9	1545.0	**1563.6**	16.4	0.5
101-120	1381.3	1361.6	1221.7	**1275.6**	36.7	4.8
121-140	2200.4	2217.4	2159.3	**2191.8**	23.9	1.0
141-160	2048.7	2275.9	1882.2	**2001.4**	80.0	4.7
i325						
1-20	759.4	768.6	748.9	**751.3**	3.6	0.0
21-40	1222.6	1251.1	1207.4	**1213.4**	8.0	0.9
41-60	**1346.4**	1364.3	1335.3	1357.2	24.3	0.1
61-80	1818.5	1818.6	1784.9	**1793.2**	9.6	1.9
81-100	2018.1	**2004.7**	1986.2	2006.6	22.8	0.3
101-120	2371.9	2342.2	2288.6	**2318.9**	26.4	3.0
121-140	2572.6	**2560.8**	2546.0	2567.4	30.3	0.3
141-160	2976.5	2947.7	2888.3	**2943.1**	41.6	4.3
i605						
1-20	1053.2	1037.2	969.6	**979.7**	10.2	0.1
21-40	1378.0	1642.4	1175.0	**1207.8**	15.5	5.1
41-60	1672.1	1699.9	1608.4	**1627.6**	18.4	0.4
61-80	2179.1	2515.9	1880.4	**1935.0**	25.6	6.9
81-100	2240.5	2271.2	2162.9	**2186.5**	18.7	1.0
101-120	2704.5	3271.9	2437.7	**2502.4**	35.6	7.3
121-140	2809.0	2847.4	2718.7	**2755.4**	20.2	1.5
141-160	3394.5	4220.6	3049.4	**3150.4**	51.4	7.3
i615						
1-20	1021.0	1013.7	956.1	**964.0**	8.7	0.1
21-40	1183.2	1254.5	981.9	**1011.6**	14.5	4.8
41-60	1584.3	1603.3	1539.8	**1553.5**	11.8	0.4
61-80	1888.2	2040.8	1625.2	**1672.7**	22.2	7.2
81-100	2231.0	2237.6	2179.6	**2209.0**	20.0	0.8
101-120	2572.8	2839.4	2310.0	**2390.0**	41.7	6.8
121-140	2791.3	2802.8	2724.7	**2753.6**	21.6	1.5
141-160	3184.8	3840.2	2930.1	**3034.4**	55.4	7.0
i625						
1-20	1051.5	1052.6	1012.6	**1017.9**	4.7	0.1
21-40	1238.6	1268.2	1159.6	**1168.7**	7.1	2.7
41-60	1710.8	1725.8	1682.9	**1696.5**	14.2	0.3
61-80	2044.7	1983.8	1899.4	**1926.5**	19.8	5.7
81-100	2164.0	2159.2	2123.4	**2140.8**	14.6	0.8
101-120	2667.7	2656.6	2480.3	**2540.7**	41.8	5.7
121-140	2784.0	2767.5	2730.8	**2752.4**	18.4	0.9
141-160	3417.1	3677.4	3225.5	**3326.3**	68.1	5.2

9.5. Errores relativos

El objetivo de esta sección es valorar la proximidad de la solución del algoritmo a una planificación óptima, medida como el porcentaje de error sobre el valor de dicha planificación. Sin embargo, por lo que sabemos no se conoce la solución óptima de estas instancias, y en estos casos las principales alternativas son las cotas inferiores o las estimaciones estadísticas del valor óptimo (técnica que a partir de una muestra de soluciones trata de predecir el valor de la solución óptima de la instancia). Pero por lo que sabemos tampoco existen ni cotas inferiores satisfactorias ni estimaciones estadísticas para estas instancias. En [118], Rardin y Uzsoy sugieren que en estos casos se utilicen los valores de la mejor solución conocida para cada problema (en adelante BKS, Best Known Solution). Para obtener los BKS utilizaremos los mejores valores existentes hasta el momento en la literatura, junto con los valores obtenidos por nuestros algoritmos en ejecuciones largas (10^6 iteraciones, como hemos comentado), ya que a menudo esos valores superan a las mejores soluciones que se pueden encontrar en la literatura. Por supuesto, utilizamos para los tres métodos el mismo BKS para calcular los errores relativos.

Una dificultad añadida al tratar con el maximum lateness es que los valores de la función objetivo pueden ser negativos. De nuevo hacemos lo sugerido por Rardin y Uzsoy en [118] y sumamos el mínimo due date de la instancia a los valores del maximum lateness, tanto en el numerador como en el denominador de las fracciones. La justificación de este procedimiento es que de esta forma el lateness de al menos el trabajo con mínimo due date no será negativo, y por lo tanto el maximum lateness de toda la planificación tampoco lo será. Hay que tener en cuenta que esta transformación subestimará el porcentaje de error de las soluciones heurísticas. Sin embargo, como ocurre lo mismo para los tres métodos que se consideran, las comparaciones entre ellos no deberían verse afectadas.

La tabla 9.3 resume los resultados de SB-GLS (columna 6), ISS (columna 7), y nuestro método $TS - N^S$ cuando se ejecuta por un tiempo equivalente a SB-GLS (columnas 8 y 9 para el resultado medio y el mejor, respectivamente, en 30 ejecuciones). Los resultados para las 960 instancias se promedian para cada grupo de 20 instancias, de acuerdo a los valores de los parámetros τ, R, N y M utilizados para generarlas, los cuales se indican en las primeras columnas. La razón de incluir el mejor valor y el valor medio obtenido por $TS - N^S$ es que tanto SB-GLS como ISS son también métodos aproximados, y los autores no dejan claro si los valores publicados son realmente las medias muestrales.

Tabla 9.3: Experimentos maximum lateness: Errores porcentuales medios

					SB-GLS	ISS	$TS-N^S$	
Grupo	τ	R	M	N			Media	Mejor
i305	0.3	0.5	5	10	3.95	4.80	0.59	0.06
				20	10.19	21.50	2.33	0.81
			10	10	3.72	4.44	1.11	0.08
				20	11.38	19.91	2.27	0.53
			15	10	2.81	3.55	0.85	0.11
				20	10.13	22.59	2.99	0.70
			20	10	2.71	3.02	0.77	0.12
				20	8.46	29.43	3.86	1.55
i315	0.3	1.5	5	10	2.06	1.96	0.83	0.04
				20	13.42	11.95	1.98	0.10
			10	10	1.56	2.23	0.59	0.08
				20	8.05	4.72	2.41	0.63
			15	10	2.71	2.27	0.94	0.02
				20	9.14	8.65	4.34	1.70
			20	10	1.60	2.24	1.29	0.10
				20	8.77	17.63	7.45	3.05
i325	0.3	2.5	5	10	1.36	2.48	0.30	0.40
				20	1.27	5.93	0.85	0.09
			10	10	1.20	2.36	1.94	0.18
				20	2.23	2.50	0.67	0.11
			15	10	1.62	0.86	1.13	0.01
				20	4.33	3.06	1.98	0.37
			20	10	1.08	0.63	0.86	0.05
				20	4.54	3.32	3.17	0.92
i605	0.6	0.5	5	10	5.41	4.44	0.79	0.16
				20	7.91	20.17	2.01	0.64
			10	10	3.12	4.31	1.04	0.16
				20	9.37	20.79	2.10	0.33
			15	10	2.81	3.93	0.87	0.03
				20	7.86	23.76	2.72	0.95
			20	10	2.95	4.11	1.38	0.29
				20	9.14	28.95	3.64	1.26
i615	0.6	1.5	5	10	5.22	4.60	0.64	0.00
				20	12.84	19.04	2.68	0.77
			10	10	2.54	3.57	0.81	0.08
				20	12.21	19.85	2.68	0.47
			15	10	2.34	2.65	1.47	0.29
				20	10.17	20.22	4.13	1.25
			20	10	2.33	2.71	1.09	0.13
				20	9.49	28.71	5.05	1.98
i625	0.6	2.5	5	10	3.93	4.13	0.64	0.05
				20	6.86	11.52	1.30	0.32
			10	10	1.87	2.72	0.98	0.13
				20	7.97	5.48	2.07	0.38
			15	10	2.27	2.00	1.09	0.27
				20	8.65	9.21	3.96	1.23
			20	10	2.05	1.44	0.90	0.12
				20	9.10	18.23	5.91	2.39
Media global					5.56	9.35	1.99	0.53
Peor global					47.84	48.04	13.47	6.46
Mejor global					0.00	0.00	0.00	0.00

A partir de los resultados que se muestran es claro que el algoritmo propuesto mejora a los otros dos métodos en cuanto a la calidad de la solución, ofreciendo resultados significativamente mejores incluso en el peor de los casos. Si tomamos el valor medio obtenido con $TS - N^S$, la reducción del error con respecto a SB-GLS es en media 58,5 % (32 % para un conjunto de instancias y más del 50 % para los cinco conjuntos restantes). Comparado con ISS, $TS - N^S$ reduce el error en un 65,43 % de media, siendo la reducción mayor del 75 % en tres de los seis conjuntos de instancias. La relevancia que tiene el método que proponemos sería todavía mayor si los resultados publicados en [20] y [107] se refiriesen a los mejores valores alcanzados; en ese caso, tomando los mejores valores alcanzados por $TS - N^S$, la reducción del error relativo con respecto a los otros dos métodos sería mayor del 90 %.

Para poder ver más claramente el comportamiento de nuestro método en las 960 instancias, la tabla 9.4 muestra el número de veces en las que $TS - N^S$ obtiene mejores, iguales o peores resultados que SB-GLS e ISS. Observamos que el mejor valor y el valor medio obtenido por $TS - N^S$ mejora a los otros algoritmos en 87.4–90.5 % y 80.0–84.0 % de las instancias, respectivamente.

Tabla 9.4: Experimentos maximum lateness: Comparación entre SB-GLS, ISS y $TS - N^S$ sobre las 960 instancias

$TS - N^S$	Mejor vs.		Media vs.	
	SB-GLS	ISS	SB-GLS	ISS
Mejor	839	869	769	809
Igual	113	86	57	45
Peor	8	5	134	106

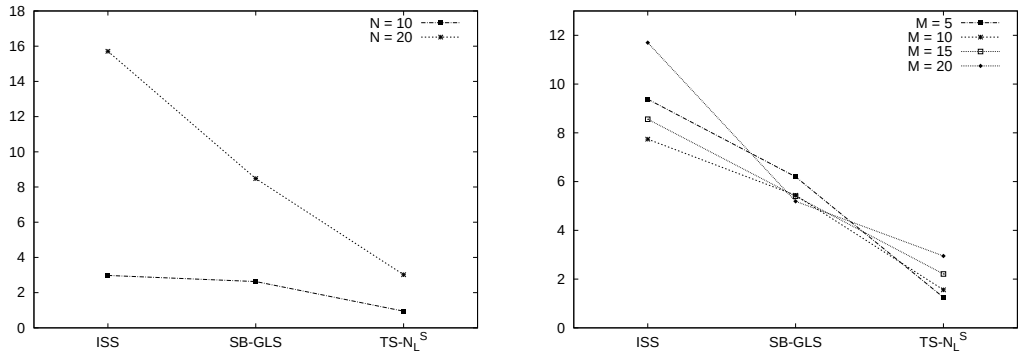
Además, hemos calculado los intervalos de confianza al 95 % para los errores relativos obtenidos por los tres algoritmos. Recordemos que el límite inferior de dicho intervalo se calcula como la media menos 1.96 veces el error estándar de la media y el límite superior como la media más 1.96 veces el error estándar de la media. Entonces, la cota superior del intervalo para $TS - N^S$ es $1{,}99 + 0{,}12 = 2{,}11$, y es claramente menor que la cota inferior del intervalo para ISS, que es $9{,}35 - 0{,}61 = 8{,}74$, y para SB-GLS, que es $5{,}56 - 0{,}32 = 5{,}24$.

9.6. Sensibilidad a los parámetros que definen las instancias

La tabla 9.5 ofrece más detalle sobre el efecto que tienen los parámetros que definen las instancias (N, M, R y τ) en cada uno de los algoritmos, mostrando los errores medios para cada uno de los valores posibles de los parámetros. Por la misma razón que en la sección anterior, mostramos también los mejores resultados alcanzados por $TS - N^S$, junto con el peor resultado observado en cualquier instancia para cada combinación de parámetros. El mejor resultado es siempre igual a 0.00, debido a que siempre existe al menos una instancia en cada grupo para la que cada algoritmo obtiene un maximum lateness igual a la mejor solución conocida.

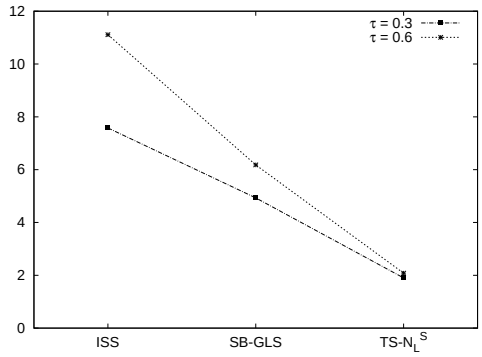
Tabla 9.5: Experimentos maximum lateness: Errores porcentuales de cada método según los parámetros de la instancia

		ISS	SB-GLS	Media TS	Mejor TS
$N = 10$	Media	2.98	2.63	0.95	0.12
	Peor	12.77	18.09	9.11	4.41
$N = 20$	Media	15.71	8.48	3.02	0.94
	Peor	48.04	47.84	13.47	6.46
$M = 5$	Media	9.38	6.20	1.25	0.29
	Peor	48.04	47.84	7.75	3.58
$M = 10$	Media	7.74	5.44	1.56	0.26
	Peor	29.86	21.38	9.11	3.64
$M = 15$	Media	8.56	5.40	2.21	0.58
	Peor	36.42	22.21	13.47	5.47
$M = 20$	Media	11.70	5.19	2.95	1.00
	Peor	40.21	17.81	12.06	6.46
$R = 0{,}5$	Media	13.73	6.37	1.83	0.49
	Peor	35.73	17.14	5.86	4.41
$R = 1{,}5$	Media	9.56	6.53	2.40	0.67
	Peor	48.04	47.84	13.47	6.46
$R = 2{,}5$	Media	4.74	3.77	1.73	0.44
	Peor	38.34	24.62	9.53	5.13
$\tau = 0{,}3$	Media	7.58	4.93	1.90	0.49
	Peor	48.04	47.84	13.47	6.46
$\tau = 0{,}6$	Media	11.11	6.18	2.08	0.57
	Peor	40.21	24.62	9.05	5.13

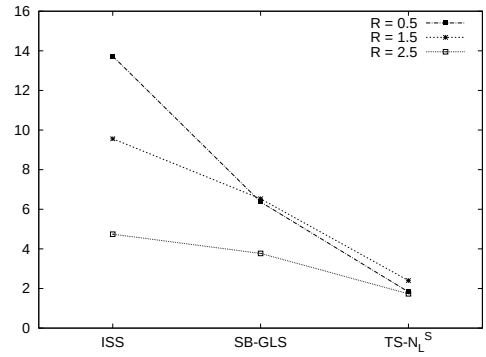


(a) Número de trabajos (N)

(b) Número de máquinas (M)



(c) Tardiness (τ)



(d) Spread (R)

Figura 9.1: Experimentos maximum lateness: Interacción entre parámetros y algoritmos, con el error respecto a la mejor solución conocida en el eje Y

Claramente, los valores de los parámetros tienen un impacto considerablemente menor en $TS-N^S$ que en SB-GLS e ISS. Esto también se puede apreciar en la figura 9.1. En cuanto al número de máquinas M, $TS-N^S$ obtiene los mejores resultados para $M = 5$. Éste también es el caso para la mayor reducción del error con respecto a los otros dos métodos: 87 % menos que ISS y 80 % menos que SB-GLS. Para $TS-N^S$ el error se incrementa con el número de máquinas, lo cual no ocurre con SB-GLS.

En cuanto al número de trabajos N, todos los algoritmos obtienen los errores más pequeños para $N = 10$. $TS-N^S$ obtiene una reducción del error medio con respecto a ISS del 74 %, alcanzando 81 % para el mayor número de trabajos. Comparándolo con SB-GLS, el error se reduce en un 64 % tanto con $N = 10$ como con $N = 20$. También se puede observar que la variación del error para cambios en el número de máquinas es menos significativa que para cambios en el número de trabajos.

Para el tardiness τ, todos los algoritmos obtienen las menores tasas de error con $\tau = 0{,}3$, lo que sugiere que la dificultad de las instancias aumenta con el número de trabajos que se espera que acaben más tarde de su due date. La mayor reducción en la tasa de error obtenida por $TS - N^S$ se consigue cuando las instancias parecen ser más difíciles ($\tau = 0{,}6$): 81 % con respecto a ISS y 66 % con respecto a SB-GLS.

Finalmente, sobre el parámetro spread R, todos los métodos obtienen las menores tasas de error medio para $R = 2{,}5$, lo que sugiere que las instancias son más fáciles si la amplitud de los due dates es muy grande. La mejora de $TS - N^S$ para los restantes valores de R es clara, con una reducción del error de $75 - 87$ % con respecto a ISS y de $63 - 71$ % con respecto a SB-GLS.

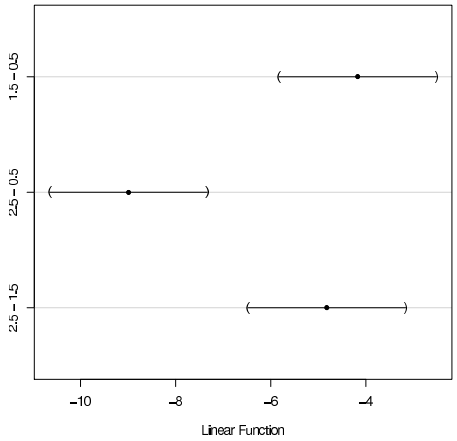
En todos los casos, el análisis utiliza como referencia el valor del error medio para $TS - N^S$, no el mejor valor obtenido; obviamente, la diferencia en los resultados sería mayor si comparásemos nuestros mejores valores.

Además, hemos realizado varios tests estadísticos para comprobar la interacción entre parámetros y algoritmos. Hemos aplicado los tests Shapiro-Wilks, Barlett, t-test y ANOVA. Cuando las hipótesis de normalidad u homocedasticidad necesarias para aplicar ANOVA no se cumplen, se utiliza en su lugar el test no paramétrico de Kruskal-Wallis. Tras un análisis estadístico previo, se aplican también los tests de Tukey para analizar las interacciones.

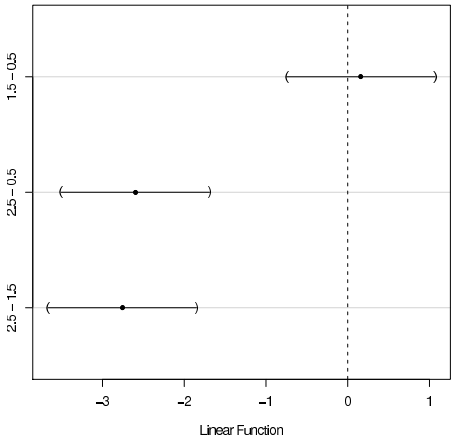
Como el parámetro τ sólo toma dos valores, se ejecuta un t-test para cada método. Los t-test indican la existencia de diferencias estadísticamente significativas para cada valor de τ (p-value=0.00) excepto en el algoritmo $TS - N^S$, para el que no es evidente asumir diferencias entre valores de τ (p-value=0.14).

En cuanto al número de máquinas M, hemos aplicado el test Kruskal-Wallis a todos los algoritmos, ya que no podemos asumir normalidad (el test Shapiro-Wilks da un p-valor $<$0.05) ni homocedasticidad (el test Barlett da un p-valor $<$0.05). Para ISS, se rechaza la existencia de diferencias para cada valor de M (ANOVA da un p-valor=0.00 indicando que existen interacciones). También se rechaza la influencia de M en SB-GLS, mientras que para $TS - N^S$ ambos tests, ANOVA y Kruskal-Wallis (p-valor=0.00) indican que la probabilidad de que el valor de M no tenga efecto en el error medio es prácticamente nula. Además, el test Tukey muestra que hay diferencias en el error medio para cada par de valores, excepto para 10 y 5 máquinas (p-valor=0.25), donde $TS - N^S$ obtiene los mejores resultados y las mayores reducciones del error con respecto a los otros dos métodos.

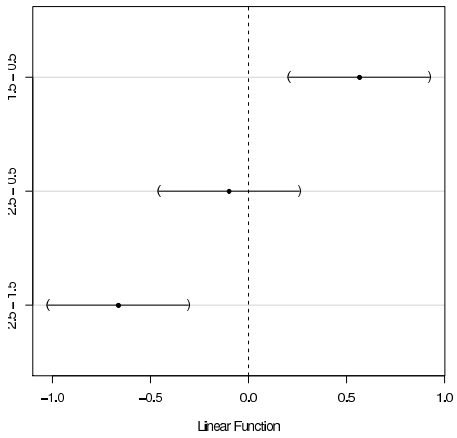
Para el número de trabajos N, como sólo hay dos valores diferentes, realizamos un único



(a) ISS



(b) SB-GLS



(c) $TS - N^S$

Figura 9.2: Experimentos maximum lateness: Intervalos de confianza del análisis de pares de valores de R para los tres métodos

t-test. Los resultados muestran que en todos los casos existen diferencias entre ambos valores.

Finalmente, para el parámetro R, los tests de Kruskal-Wallis muestran que para todos los algoritmos hay diferencias en los errores relativos obtenidos para cada grupo de instancias. También se han aplicado los tests Tukey, con la conclusión de que hay diferencias entre los valores medios de los errores relativos, con sólo dos excepciones: el par 2,5−0,5 para $TS-N^S$ (donde este método obtiene las tasas medias de error más bajas) y el par $1{,}5-0{,}5$ para SB-GLS. La figura 9.2 muestra los intervalos de confianza del análisis de pares de valores de R para (a) ISS, (b) SB-GLS, y (c) $TS - N^S$, mejorando el gráfico de la figura 9.1(d) (ver las

diferencias en los valores del eje X).

Como conclusión, el método propuesto es a la vez eficiente y robusto con respecto a los diferentes parámetros utilizados para generar las instancias. Para todas las configuraciones de parámetros, $TS - N^S$ ofrece el mejor comportamiento. Por otra parte ISS parece ser el algoritmo más variable, aunque en el peor caso es SB-GLS el que tiene mayor dependencia en los parámetros de la instancia.

9.7. Ejecuciones cortas frente a ejecuciones largas

Los valores de desviación estándar indicados en la columna 6 de la tabla 9.2 de la sección 9.4 sugieren que hay posibilidades de mejorar los resultados obtenidos por $TS - N^S$ si se le deja un tiempo de ejecución mayor. Esto motiva la segunda serie de experimentos, en la cual dejamos correr el algoritmo durante 10^6 iteraciones para cada una de las instancias, para evaluar su capacidad de mejorar con tiempos de ejecución más elevados. Los resultados se muestran en las columnas 6 y 7 de la tabla 9.6. Como se podía esperar, si comparamos los resultados obtenidos por $TS - N^S$ en la primera y segunda series de experimentos, se puede ver una clara mejora utilizando un mayor número de iteraciones, tanto en la mejor solución como en la solución media. En cuanto a la comparación con el resto de métodos, la solución media alcanzada por $TS - N^S$ es ligeramente peor que SB-GLS únicamente en las instancias $41 - 60$ del benchmark i325 y es mejor que ISS en todos y cada uno de los 48 grupos. En este caso, la cota superior del intervalo de confianza al 95 % es menor que los valores de SB-GLS e ISS en el 91,7 % y 93,8 % de los grupos, respectivamente.

Tabla 9.6: Experimentos maximum lateness: Ejecuciones cortas frente a largas

Inst.	SB-GLS	ISS	$TS-N^S$ Mejor	$TS-N^S$ Media	$TS-N^S$ 10^6 iter. Mejor	$TS-N^S$ 10^6 iter. Media
i305						
1-20	361.7	372.2	305.4	313.0	304.5	**304.5**
21-40	93.9	294.3	-122.2	-90.9	-137.9	**-114.7**
41-60	1087.1	1102.3	1010.9	1032.4	1009.1	**1010.1**
61-80	815.2	1041.0	526.6	572.6	513.2	**543.1**
81-100	1650.7	1670.4	1577.5	1597.5	1574.6	**1577.0**
101-120	1521.0	1933.8	1205.3	1281.2	1184.4	**1219.6**
121-140	2158.9	2170.0	2077.0	2097.9	2073.2	**2078.3**
141-160	2097.0	2908.1	1815.4	1906.1	1755.1	**1801.7**
i315						
1-20	425.2	425.0	407.6	414.9	408.3	**410.0**
21-40	202.6	185.2	67.5	85.9	66.7	**79.6**
41-60	1081.5	1094.8	1060.0	1068.3	1058.7	**1060.1**
61-80	757.4	679.9	621.1	646.8	611.4	**628.5**
81-100	1597.5	1589.9	1545.0	1563.6	1547.2	**1549.6**
101-120	1381.3	1361.6	1221.7	1275.6	1187.3	**1209.5**
121-140	2200.4	2217.4	2159.3	2191.8	2156.6	**2160.7**
141-160	2048.7	2275.9	1882.2	2001.4	1799.1	**1831.6**
i325						
1-20	759.4	768.6	748.9	751.3	748.5	**749.9**
21-40	1222.6	1251.1	1207.4	**1213.4**	1206.6	**1213.4**
41-60	**1346.4**	1364.3	1335.3	1357.2	1337.0	1350.0
61-80	1818.5	1818.6	1784.9	1793.2	1783.3	**1788.4**
81-100	2018.1	2004.7	1986.2	2006.6	1985.9	**1999.2**
101-120	2371.9	2342.2	2288.6	2318.9	2281.4	**2290.8**
121-140	2572.6	2560.8	2546.0	2567.4	2544.6	**2551.2**
141-160	2976.5	2947.7	2888.3	2943.1	2866.6	**2891.2**
i605						
1-20	1053.2	1037.2	969.6	979.7	967.0	**967.0**
21-40	1378.0	1642.4	1175.0	1207.8	1160.0	**1183.7**
41-60	1672.1	1699.9	1608.4	1627.6	1604.7	**1607.1**
61-80	2179.1	2515.9	1880.4	1935.0	1872.6	**1901.7**
81-100	2240.5	2271.2	2162.9	2186.5	2162.0	**2163.9**
101-120	2704.5	3271.9	2437.7	2502.4	2403.0	**2443.4**
121-140	2809.0	2847.4	2718.7	2755.4	2717.4	**2723.6**
141-160	3394.5	4220.6	3049.4	3150.4	2996.9	**3045.4**
i615						
1-20	1021.0	1013.7	956.1	964.0	956.1	**956.1**
21-40	1183.2	1254.5	981.9	1011.6	971.8	**990.4**
41-60	1584.3	1603.3	1539.8	1553.5	1538.2	**1539.5**
61-80	1888.2	2040.8	1625.2	1672.7	1618.2	**1646.8**
81-100	2231.0	2237.6	2179.6	2209.0	2172.4	**2179.3**
101-120	2572.8	2839.4	2310.0	2390.0	2276.7	**2312.4**
121-140	2791.3	2802.8	2724.7	2753.6	2720.5	**2726.3**
141-160	3184.8	3840.2	2930.1	3034.4	2862.6	**2910.7**
i625						
1-20	1051.5	1052.6	1012.6	1017.9	1012.1	**1013.1**
21-40	1238.6	1268.2	1159.6	1168.7	1156.7	**1163.3**
41-60	1710.8	1725.8	1682.9	1696.5	1680.5	**1681.9**
61-80	2044.7	1983.8	1899.4	1926.5	1893.6	**1907.3**
81-100	2164.0	2159.2	2123.4	2140.8	2117.4	**2123.1**
101-120	2667.7	2656.6	2480.3	2540.7	2452.1	**2477.1**
121-140	2784.0	2767.5	2730.8	2752.4	2727.3	**2734.0**
141-160	3417.1	3677.4	3225.5	3326.3	3157.3	**3185.4**

Por otra parte, la tabla 9.7 contiene un resumen de los resultados obtenidos en las ejecuciones cortas (es decir en condiciones de ejecución similares a SB-GLS) y en las ejecuciones largas (con 10^6 iteraciones). Los errores relativos de las 30 ejecuciones de nuestro algoritmo de búsqueda tabú se promedian para cada uno de los seis conjuntos de instancias. También incluímos en cada caso la desviación estándar del error relativo medio y el tiempo medio de una única ejecución.

Tabla 9.7: Experimentos maximum lateness: Resumen de las ejecuciones cortas frente a las ejecuciones largas

	Cortas		Largas	
Benchmark	Media±Desv	T(s)	Media±Desv	T(s)
i305	1.85±1.26	3.38	0.62±0.62	58.08
i315	2.48±2.74	2.38	0.71±1.07	41.12
i325	1.36±1.96	1.36	0.68±1.44	27.21
i605	1.82±1.18	3.69	0.61±0.64	60.10
i615	2.32±1.75	3.55	0.75±0.77	54.13
i625	2.11±2.17	2.67	0.56±0.71	43.51

Ya comentamos que los valores de la desviación estándar en los experimentos cortos invitan a probar con tiempos de ejecución más elevados, para comprobar si los resultados pueden mejorar más. Esto se confirma viendo que los valores medios y las desviaciones estándar de las ejecuciones largas son claramente mejores, lo que indica que el incremento en el número de iteraciones no sólo consigue mejores resultados, sino que además consigue resultados más estables.

Por último, para ver en detalle resultados de ejecuciones cortas frente a ejecuciones largas en instancias concretas, en la tabla 9.8 se resumen los resultados obtenidos en las primeras cinco instancias de los cuatro grupos con 20 trabajos del benchmark i305, tal y como se hace en [20]. La tabla muestra, para cada una de las 20 instancias, la mejor solución ofrecida en [20], el maximum lateness medio de una ejecución corta (Media(c)), de una ejecución larga (Media(l)), y la mejor solución alcanzada por TS (Mejor-TS). De nuevo se puede apreciar como el maximum lateness mejora cuando se le deja a TS un número de iteraciones mayor. Además también observamos que en todas las instancias, tanto la media de las ejecuciones cortas como la de las ejecuciones largas mejora notablemente los resultados alcanzados por SB-GLS.

Tabla 9.8: Experimentos maximum lateness: Ilustrando los beneficios de una búsqueda tabú más larga en instancias concretas del benchmark i305

Instancia	Tamaño	SB-GLS	Media(c)	Media(l)	Mejor-TS
21		493	295.6	266.5	246
22		-158	-310.9	-335.0	-367
23	20×5	156	-64.4	-87.1	-106
24		-81	-164.5	-184.8	-226
25		-64	-335.1	-362.1	-368
61		548	404.9	385.3	344
62		707	479.6	441.4	403
63	20×10	1035	770.3	750.3	728
64		988	677.9	651.0	630
65		553	344.1	316.8	293
101		1511	1368.6	1315.6	1284
102		1459	1337.6	1269.9	1237
103	20×15	1487	1285.1	1225.7	1186
104		1934	1609.3	1535.6	1505
105		1756	1470.7	1393.8	1331
141		1871	1665.9	1604.0	1575
142		1902	1754.5	1666.4	1618
143	20×20	2223	2022.8	1930.3	1885
144		2092	1861.1	1778.1	1738
145		1987	1776.1	1691.1	1633

9.8. Conclusiones

A partir del estudio realizado podemos concluir que nuestro algoritmo $TS - N^S$ mejora al método SB-GLS propuesto por Balas et al. en [20] utilizando tiempos de ejecución equivalentes. También mejora los resultados del método ISS propuesto por Oddi et al. en [107], pero en este caso tenemos que tener en cuenta las diferencias entre las condiciones de ejecución. Además, el análisis de la influencia de los parámetros utilizados para generar las instancias muestra la robustez del método propuesto ante los cambios en dichos parámetros. También hemos comprobado que $TS - N^S$ puede mejorar todavía más las soluciones obtenidas aumentando las iteraciones de la búsqueda tabú, aunque con un tiempo de ejecución mayor tal vez los resultados serían aún mejores combinándo la búsqueda tabú con un algoritmo genético. Además, mejoramos la mejor solución conocida en 817 de las 960 instancias del benchmark utilizado, lo que puede servir como referencia a futuros trabajos.

Capítulo 10

ESTUDIO EXPERIMENTAL. MINIMIZACIÓN DEL WEIGHTED TARDINESS

10.1. Introducción

El objetivo de este estudio experimental es comparar nuestros métodos de minimización del weighted tardiness con otros algoritmos del estado del arte. El primer paso es optimizar los parámetros utilizados para tratar con esta función objetivo, por lo que realizamos primero un ajuste de parámetros, relacionado con el uso de estimaciones, la estructura de vecindad utilizada y el número de caminos críticos considerados. A continuación hemos comparado nuestro algoritmo con métodos que minimizan el weighted tardiness en instancias del JSP clásico sin tiempos de setup, y posteriormente mostraremos resultados sobre dos conjuntos de instancias del SDST-JSP. Sin embargo, el único algoritmo que conocemos que minimice esta función objetivo en el SDST-JSP es el shifting bottleneck propuesto por Sun y Noble en [139]. El problema es que en su estudio experimental utilizan instancias generadas de forma aleatoria, y aunque hemos contactado con los autores, ya no disponen ni del código de su algoritmo ni de las instancias utilizadas en su estudio. Sabemos los parámetros generales utilizados para construirlas, pero debido a que el número de instancias que utilizan para

cada posible configuración de dichos parámetros es muy reducido, la aleatoriedad jugaría un papel demasiado importante y la comparación no podría ser en absoluto precisa. Por este motivo hemos comparado nuestros resultados con los obtenidos por el método general ILOG CPLEX CP Optimizer.

10.2. Asignación de due dates y pesos a los trabajos

Cuando en una instancia, ya sea del JSP o del SDST-JSP, no se definan los due dates y los pesos de cada trabajo, los asignaremos con el mismo método utilizado por otros investigadores en la literatura, entre otros por Kreipl en [79] y por Essafi et al. en [49]. Los pesos de cada trabajo y sus due dates se asignarán de la siguiente forma: al primer 20 % de los trabajos se les asigna un peso 4 (trabajos muy importantes), al siguiente 60 % de los trabajos se les asigna un peso 2 (importancia media) y a los restantes trabajos se les asigna peso 1 (poco importantes). Por ejemplo, si la instancia tiene 10 trabajos, entonces $w_1 = w_2 = 4$, $w_3 = \cdots = w_8 = 2$ y $w_9 = w_{10} = 1$. Esta forma de asignar pesos no es completamente arbitraria, sino que esta basada en observaciones en entornos de producción reales. Además a cada trabajo i se le asigna un due date d_i de acuerdo a la siguiente fórmula:

$$d_i = f * \sum_{j=1}^{M} p_{ij}, \tag{10.1}$$

donde M es el número total de máquinas del problema, o lo que es equivalente, el número total de tareas del trabajo i. Es decir, el due date d_i de un trabajo i es la suma de los tiempos de procesamiento de todas sus tareas, multiplicado por un factor f. Mediante ese parámetro f se puede controlar lo estrictos que son esos due dates. En la literatura se suelen utilizar los valores $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$, que en instancias cuadradas del JSP (es decir con igual número de trabajos que de máquinas) corresponden a due dates muy estrictos, moderadamente estrictos, y poco estrictos.

10.3. Minimización del weighted tardiness en el JSP

Para comparar nuestro algoritmo con otros, consideraremos los mejores métodos conocidos en el estado de arte, que por lo que sabemos, son el algoritmo de ramificación y poda propuesto por Singer y Pinedo en [132], el algoritmo de búsqueda local de tipo large step

random walk (LSRW) propuesto por Kreipl en [79], el algoritmo genético híbrido propuesto por Essafi et al. en [49] y el método de búsqueda local propuesto por Mati et al. en [86].

La búsqueda local propuesta en [86] en realidad no esta dedicada en exclusiva a la minimización del weighted tardiness, sino que su método es capaz de minimizar cualquier función objetivo regular, como ya hemos comentado en la sección 6.3.1 de esta tesis. Sin embargo, incluímos en este capítulo los resultados obtenidos por su método en la minimización del weighted tardiness por considerarlos de muy buena calidad.

Otros dos algoritmos del estado del arte son el algoritmo genético híbrido propuesto por Zhou et al. en [164] y el heurístico shifting bottleneck propuesto por Singer y Pinedo en [133], sin embargo omitimos los resultados de estos dos métodos debido a que obtienen peores resultados que los demás métodos citados. En [146], Van Hentenryck y Michel también presentan resultados de su método de búsqueda local sobre la minimización del weighted tardiness, aunque no con la intención de compararse con gran detalle con otros métodos del estado del arte, sino más bien con la intención de demostrar la eficacia de las abstracciones que proponen en dicho trabajo. Los valores que obtienen son competitivos pero no son mejores que los que se presentan en [49] o en [86]. Aunque no indicaremos sus resultados en la tabla de datos correspondiente por no incluir demasiados algoritmos en dicha tabla, en el texto realizaremos una breve comparación con su método.

El primer conjunto de instancias que utilizaremos fue originalmente propuesto por Singer y Pinedo en [132], y se compone de 22 instancias de tamaño 10×10, en concreto: ABZ5, ABZ6, LA16 a LA24, MT10, y ORB1 a ORB10. Hay que tener en cuenta que en realidad las instancias LA21 a LA24 son de tamaño 15×10, y para convertirlas al tamaño 10×10 el método utilizado es quitar los últimos 5 trabajos de la instancia. Los due dates y los pesos se asignan utilizando el método detallado en la sección 10.2, y considerando los valores $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$. Éste es el benchmark más utilizado en la literatura sobre minimización del weighted tardiness en el JSP, y por este motivo nos compararemos en él con todos los demás algoritmos.

El segundo conjunto de instancias fue propuesto por Essafi et al. en [49], con instancias de un tamaño en general mayor y con el propósito de compararse con otros métodos en el futuro. El tamaño de las instancias varía desde 10×5 para las más pequeñas hasta 30×10 y 15×15 para las más grandes. El conjunto se basa en las instancias propuestas originalmente por Lawrence en [82], utilizadas en gran número de trabajos como benchmarks para la minimización del makespan. Los due dates y los pesos se asignan de nuevo utilizando el

método descrito en la sección 10.2, y se consideran los valores $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$. Observemos que las instancias LA16 a LA20 ya se tratan en el primer conjunto de instancias. En cambio las instancias LA21 a LA24 no se trataron, porque recordemos que en el primer conjunto de instancias se recortan para reducir su tamaño a 10×10. En este conjunto de instancias nos compararemos únicamente con el método propuesto en [49], ya que no disponemos de resultados del resto de métodos sobre este benchmark.

Sin embargo, antes de compararnos con los mejores métodos conocidos realizaremos un análisis paramétrico previo, para comprobar si en las funciones objetivo de tipo suma es conveniente cambiar algunos aspectos del algoritmo respecto de la configuración más eficiente que habíamos encontrado para las funciones objetivo de tipo bottleneck. Concretamente revisaremos la utilización de estimaciones, el tipo de estructura de vecindad, el número de caminos críticos que se considerarán, y si el híbrido de algoritmo genético y búsqueda tabú sigue siendo la mejor opción. Para realizar este análisis utilizamos como benchmark el primer conjunto de instancias ya que, aunque todas sean del mismo tamaño, es el conjunto más ampliamente utilizado en la literatura sobre minimización del weighted tardiness y por lo tanto tiene mucho interés. Por otra parte utilizamos el factor $f = 1{,}3$, es decir los due dates más estrictos, ya que cuanto más difícil sea el cumplimiento de los due dates, mayor dificultad suele tener la resolución de las instancias.

10.3.1. Utilización de estimaciones

Como hemos comprobado en la sección 8.3.9, la mejor opción para las funciones de tipo bottleneck es estimar la función objetivo de todos los vecinos generados, y elegir al mejor de todos ellos. Esto tenía mucho sentido ya que en la sección 7.6.5 hemos visto que el algoritmo de estimación es muy preciso para ese tipo de funciones. Sin embargo, en esa misma sección también comprobamos que la estimación para las funciones objetivo de tipo suma es mucho menos precisa.

Por este motivo, en esta sección compararemos las dos siguientes opciones: elegir de entre toda la vecindad el individuo con mejor estimación, o calcular de forma exacta la función objetivo para todos los individuos cuya estimación sea menor que la función objetivo del individuo original, y posteriormente elegir al mejor individuo. Los vecinos con una estimación mayor que la función objetivo del individuo original no los tenemos en cuenta porque en la sección 7.6.5 hemos comprobado que la estimación es mayor que la función objetivo real del individuo muy rara vez, y por ello el gran aumento de tiempo de ejecución no puede

compensar tenerlos en cuenta.

Para comparar estas dos opciones experimentalmente hemos realizado las pruebas cuyos resultados mostramos en la tabla 10.1. Como ya hemos comentado utilizamos como benchmark el primer conjunto de instancias con el factor $f = 1{,}3$. Comenzaremos con un híbrido de algoritmo genético y búsqueda tabú con la estructura de vecindad N^S, ya que es el método que mejores resultados consiguió en el capítulo 8.

Tabla 10.1: Experimentos weighted tardiness: Diferentes formas de utilizar el algoritmo de estimación

	Sólo mejor estim.		*Estim.* $<WT$	
Inst.	Mejor	Media	Mejor	Media
ABZ5	1403(7)	1486.9	1403(29)	**1403.2**
ABZ6	436(6)	503.2	436(30)	**436.0**
LA16	1169(8)	1238.9	1169(28)	**1172.9**
LA17	899(9)	1041.4	899(26)	**941.2**
LA18	929(29)	930.8	929(30)	**929.0**
LA19	1071(1)	1144.0	948(6)	**1051.0**
LA20	846(18)	867.2	805(5)	**844.0**
LA21	463(30)	**463.0**	463(30)	**463.0**
LA22	1182(1)	1307.0	1064(4)	**1135.3**
LA23	835(2)	899.8	835(2)	**871.9**
LA24	835(24)	843.6	835(29)	**836.4**
MT10	1363(3)	1649.9	1363(23)	**1412.1**
ORB1	2568(1)	2752.5	2568(11)	**2629.9**
ORB2	1434(19)	1484.0	1408(1)	**1432.8**
ORB3	2111(1)	2314.7	2111(16)	**2162.5**
ORB4	1623(3)	1729.2	1623(24)	**1629.3**
ORB5	1667(12)	1730.0	1593(4)	**1663.0**
ORB6	1790(7)	1974.7	1790(25)	**1807.0**
ORB7	590(6)	613.7	590(24)	**596.6**
ORB8	2429(4)	2561.6	2429(10)	**2457.9**
ORB9	1316(14)	1414.3	1316(28)	**1326.9**
ORB10	1815(1)	1979.7	1679(19)	**1721.4**

Elegimos la configuración con el objetivo de que el tiempo medio de una única ejecución esté comprendido entre 17 y 19 segundos, es decir el mismo tiempo de ejecución utilizado tanto por Essafi et al. en [49] como por Mati et al. en [86], ya que las máquinas utilizadas en esta tesis y en esos trabajos son comparables. El algoritmo propuesto por Essafi et al.

se ejecuta en un PC con un procesador de 2.8 GHz y 512 MB de RAM, el propuesto por Mati et al. se ejecuta en un Pentium con un procesador de 2.6 GHz, y el nuestro se ejecuta en un PC con un procesador Intel Core 2 Duo a 2.66GHz con 2Gb de RAM. En esta serie de experimentos, para lograr estos tiempos utilizamos una población de 34 individuos y 46 generaciones en el caso de comprobar todos los vecinos, y una población de 42 individuos y 54 generaciones en el caso de confiar plenamente en la estimación. En ambos casos continuamos la búsqueda tabú de cada cromosoma hasta que transcurran 50 iteraciones sin conseguir mejora.

Hemos marcado en negrita los casos en los que la media obtenida por un método es igual o mejor que la media obtenida por el otro. Observamos que es mucho más eficiente en todos los casos comprobar todos los vecinos cuya estimación del weighted tardiness es menor que el weighted tardiness del individuo original, a pesar de que en este caso podamos utilizar un menor número de individuos en la población y un menor número de generaciones debido al mayor consumo de tiempo de ejecución. De todas formas la diferencia en los parámetros de la ejecución para obtener tiempos de ejecución idénticos no es muy elevada, 34/46 contra 42/54. Por lo tanto éste será el método utilizado a partir de este momento.

10.3.2. Reducción del número de vecinos

Como hemos visto en la sección anterior, para conseguir los mejores resultados es necesario calcular de forma exacta el weighted tardiness de un número de vecinos que puede ser bastante elevado. Esto conlleva a que para mantener el tiempo de ejecución alrededor de 18 segundos por ejecución, el tamaño de la población y el número de generaciones utilizado es bastante reducido. Por estos motivos pensamos que quizás sea beneficioso reducir el tamaño de la vecindad para poder aumentar los parámetros de la ejecución.

Dos posibles formas de reducir el tamaño de la vecindad son utilizar una estructura de vecindad más simple, o considerar un único camino crítico en lugar de todos los existentes. En esta sección comparamos la estructura de vecindad N^S con N_1^S, ambas definidas en el capítulo 7. Además, las compararemos tanto utilizando todos los caminos críticos de la planificación, como utilizando únicamente el más relevante, es decir el que más aporta al weighted tardiness total de la planificación.

La tabla 10.2 muestra los resultados de estos experimentos. Los parámetros utilizados en los diferentes experimentos son los siguientes: para N^S utilizamos una población de 34 individuos y 46 generaciones cuando consideramos todos los caminos críticos, y una

población de 50 individuos y 62 generaciones cuando consideramos un único camino crítico, mientras que para N_1^S utilizamos una población de 50 individuos y 60 generaciones cuando consideramos todos los caminos críticos, y una población de 70 individuos y 85 generaciones cuando consideramos un único camino crítico. En todos los casos la condición de parada de la búsqueda tabú son 50 iteraciones consecutivas sin mejora. Ajustando los parámetros de esta forma, el tiempo medio de una única ejecución, para todas las configuraciones, se sitúa entre 17 y 19 segundos. En la tabla hemos marcado en negrita las medias de weighted tardiness que son iguales o mejores al del resto de configuraciones.

Tabla 10.2: Experimentos weighted tardiness: Reducción del número de vecinos (I)

	N_1^S Uno		N_1^S Todos		N^S Uno		N^S Todos	
Inst.	Mejor	Media	Mejor	Media	Mejor	Media	Mejor	Media
ABZ5	1403(1)	1479.0	1403(29)	1405.0	1409(1)	1478.9	1403(29)	**1403.2**
ABZ6	436(30)	**436.0**	436(30)	**436.0**	436(24)	449.2	436(30)	**436.0**
LA16	1169(30)	**1169.0**	1169(26)	1176.9	1169(30)	**1169.0**	1169(28)	1172.9
LA17	899(30)	**899.0**	899(22)	960.1	899(30)	**899.0**	899(26)	941.2
LA18	929(30)	929.0	929(30)	929.0	929(30)	929.0	929(30)	929.0
LA19	948(25)	**964.6**	948(3)	1032.7	948(20)	982.3	948(6)	1051.0
LA20	805(7)	**836.4**	805(8)	836.7	805(2)	843.3	805(5)	844.0
LA21	463(30)	463.0	463(30)	463.0	463(30)	463.0	463(30)	463.0
LA22	1082(29)	**1085.7**	1064(7)	1089.0	1082(21)	1093.7	1064(4)	1135.3
LA23	835(22)	845.9	835(1)	873.7	835(24)	**842.4**	835(2)	871.9
LA24	835(30)	**835.0**	835(30)	**835.0**	835(30)	**835.0**	835(29)	836.4
MT10	1363(30)	**1363.0**	1363(27)	1384.0	1363(30)	**1363.0**	1363(23)	1412.1
ORB1	2568(25)	2583.4	2568(13)	2624.0	2568(20)	**2583.3**	2568(11)	2629.9
ORB2	1408(4)	1433.1	1408(1)	1432.8	1408(11)	**1424.9**	1408(1)	1432.8
ORB3	2111(18)	**2132.9**	2111(16)	2146.6	2111(14)	2152.0	2111(16)	2162.5
ORB4	1623(3)	1729.7	1623(21)	1640.2	1652(1)	1770.6	1623(24)	**1629.3**
ORB5	1593(17)	**1632.1**	1593(5)	1657.3	1593(16)	1636.5	1593(4)	1663.0
ORB6	1790(14)	1841.2	1790(21)	1809.2	1790(12)	1830.3	1790(25)	**1807.0**
ORB7	590(3)	605.1	590(28)	**591.9**	590(2)	610.5	590(24)	596.6
ORB8	2429(1)	2487.1	2429(12)	2460.5	2439(1)	2494.9	2429(10)	**2457.9**
ORB9	1316(30)	**1316.0**	1316(29)	1321.4	1316(30)	**1316.0**	1316(28)	1326.9
ORB10	1679(22)	**1712.7**	1679(11)	1752.0	1679(19)	1732.1	1679(19)	1721.4

Para ayudarnos a tomar una decisión hemos calculado los errores porcentuales de las medias obtenidas mediante la formula $((media - BKS)/BKS) * 100$, siendo BKS la mejor solución conocida de cada instancia, que en todos los casos coincide con la mejor solución

alcanzada por las configuraciones que utilizan todos los caminos críticos. El resultado es que el error porcentual medio de N_1^S es 1.67 % cuando se utiliza un único camino crítico y 2.14 % cuando se utilizan todos, mientras que el de N^S es 2.17 % cuando se considera un camino y 2.44 % al considerarlos todos. Por lo tanto, atendiendo a las medias de weighted tardiness obtenidas, la estructura N_1^S utilizando un único camino crítico es la configuración que obtiene mejores resultados.

Sin embargo esa configuración obtiene resultados mediocres en varias instancias, por ejemplo en LA22 no es capaz de encontrar la mejor solución conocida, mientras que las configuraciones que utilizan todos los caminos críticos son capaces de encontrarla en varias ejecuciones. También observamos que en ABZ5 o en ORB8 sólo es capaz de encontrar la mejor solución conocida en 1 de las 30 ejecuciones, mientras que utilizando todos los caminos críticos se llega a ella en un buen número de casos.

La cantidad de ejecuciones que cada una de las cuatro configuraciones es capaz de obtener la mejor solución conocida es, según el orden elegido en la tabla 10.2, 402, 400, 374 y 404, del total de 660 ejecuciones (30 ejecuciones para cada una de las 22 instancias). Sin embargo observamos que las configuraciones que utilizan todos los caminos críticos son más robustas en general, ya que con cualquiera de las dos estructuras de vecindad siempre se ha conseguido obtener la mejor solución conocida en al menos una de las 30 ejecuciones en las instancias consideradas en este estudio, mientras que las configuraciones con un único camino crítico fallan en algunas instancias.

En general, parece ser que la diferencia entre utilizar un único camino crítico o todos los caminos críticos es grande. En algunas instancias, particularmente ABZ5, ORB4, ORB6, ORB7 y ORB8, el hecho de utilizar todos los caminos críticos ofrece resultados claramente mejores, se utilice la estructura de vecindad que se utilice. Sin embargo en otras instancias, particularmente LA19, LA23, ORB1 y ORB5, ofrece resultados mucho mejores utilizar un único camino crítico. En la siguiente sección propondremos una alternativa para tratar de solucionar este problema.

Elección aleatoria del número de caminos críticos que se consideran

Acabamos de comprobar experimentalmente que la mejor opción para obtener buenas medias de weighted tardiness es utilizar N_1^S con un único camino crítico, aunque para alcanzar la mejor solución conocida parece ser mejor opción utilizar todos los caminos críticos, con cualquiera de las dos estructuras de vecindad. Además, hemos visto que hay instancias

en las que nuestro algoritmo se comporta mejor considerando un único camino crítico y otras en las que se comporta mejor considerándolos todos. Debido a esto, la mejor opción podría ser una estrategia intermedia, que consistirá en que cada vez que se le vaya a aplicar la búsqueda tabú a un cromosoma, se elija aleatoriamente si en esa búsqueda tabú en concreto se van a utilizar todos los caminos críticos de la planificación, o sólo uno.

Para conseguir tiempos de ejecución equivalentes a los anteriores, los experimentos eligiendo de forma aleatoria la utilización de caminos críticos se configuran de la siguiente forma: en el caso de N^S utilizamos una población de 40 individuos y 52 generaciones, mientras que para N_1^S utilizamos una población de 58 individuos y 70 generaciones. De nuevo la condición de parada de la búsqueda tabú son 50 iteraciones consecutivas sin mejora.

La tabla 10.3 muestra resultados de los nuevos experimentos junto con los experimentos de la tabla 10.2. En el caso de esta tabla, con el propósito de mostrar la información más relevante, únicamente indicamos el número de ejecuciones en el que cada configuración es capaz de obtener la mejor solución conocida.

Tabla 10.3: Experimentos weighted tardiness: Reducción del número de vecinos (II)

		N_1^S			N^S		
Inst.	Solución	Uno	Todos	Aleat.	Uno	Todos	Aleat.
ABZ5	1403	1	29	25	0	29	22
ABZ6	436	30	30	30	24	30	30
LA16	1169	30	26	30	30	28	30
LA17	899	30	22	29	30	26	29
LA18	929	30	30	30	30	30	30
LA19	948	25	3	9	20	6	7
LA20	805	7	8	4	2	5	3
LA21	463	30	30	30	30	30	30
LA22	1064	0	7	3	0	4	3
LA23	835	22	1	3	24	2	7
LA24	835	30	30	30	30	29	29
MT10	1363	30	27	29	30	23	28
ORB1	2568	25	13	18	20	11	19
ORB2	1408	4	1	2	11	1	8
ORB3	2111	18	16	18	14	16	18
ORB4	1623	3	21	20	0	24	16
ORB5	1593	17	5	12	16	4	9
ORB6	1790	14	21	24	12	25	25
ORB7	590	3	28	28	2	24	25
ORB8	2429	1	12	17	0	10	8
ORB9	1316	30	29	30	30	28	30
ORB10	1679	22	11	25	19	19	17
Total		402	400	**446**	374	404	423

Recordemos que se ejecutan 30 pruebas para cada una de las instancias. La cantidad de ejecuciones que cada una de las nuevas configuraciones es capaz de obtener la mejor solución conocida es de 423 para N^S y de 446 para N_1^S. Recordemos que las cuatro configuraciones de la tabla 10.2 habían obtenido la mejor solución conocida en 402, 400, 374 y 404 ejecuciones. En cuanto a los valores medios, en esta tabla hemos omitido las medias de weighted tardiness para que no resultara demasiado ilegible, sin embargo incluímos dichas medias en las tablas que se presentan en las próximas secciones. El error porcentual medio de las nuevas configuraciones es de 1.73 % para N^S y de 1.36 % para N_1^S. Los errores de las cuatro configuraciones originales eran 1.67 %, 2.14 %, 2.17 % y 2.44 %.

Si consideramos ahora los resultados en instancias concretas, recordemos que la utilización de un único camino crítico daba resultados mediocres en las instancias ABZ5, ORB4, ORB6, ORB7 y ORB8. Eligiendo de forma aleatoria el número de caminos críticos vemos que los resultados en estas instancias mejoran en todos los casos a los resultados utilizando un único camino crítico. Por otra parte, la utilización de todos los caminos críticos no daba muy buenos resultados en las instancias LA19, LA23, ORB1 y ORB5. De nuevo, la elección aleatoria de los caminos críticos ha mejorado en todos los casos los resultados obtenidos en estas instancias utilizando todos los caminos críticos.

Por todos estos motivos consideramos que este nuevo método es más consistente que los anteriores, y que es capaz de combinar las buenas propiedades de las dos alternativas por separado. Elegimos finalmente la vecindad N_1^S utilizando diferente número de caminos críticos en cada cromosoma, ya que es la configuración que mejores valores ha ofrecido en esta sección.

10.3.3. Comparación con otros de nuestros métodos

En el estudio experimental sobre la minimización del makespan realizado en el capítulo 8 hemos comprobado que el método que mejores resultados ofrece es la combinación de algoritmo genético y búsqueda tabú. Sin embargo, dada la gran diferencia existente entre la minimización del makespan y del weighted tardiness, es conveniente comparar de nuevo las diferentes alternativas.

En la tabla 10.4 mostramos los resultados obtenidos por tres de los métodos propuestos en esta tesis. En las primeras columnas tenemos los mejores resultados que obtuvimos en la anterior sección sobre el algoritmo genético combinado con la búsqueda tabú con la estructura de vecindad N_1^S y utilizando un número aleatorio de caminos críticos. Las siguientes

columnas indican los resultados de la búsqueda tabú por separado, y las últimas columnas indican los resultados del algoritmo genético combinado con una búsqueda local más simple, en concreto una escalada de máximo gradiente.

Tabla 10.4: Experimentos weighted tardiness: Comparación de varios de nuestros métodos

	AG+TS		TS		AG+BL	
Inst.	Mejor	Media	Mejor	Media	Mejor	Media
ABZ5	1403(25)	1412.2	1403(19)	1409.6	1403(30)	**1403.0**
ABZ6	436(30)	**436.0**	436(20)	439.7	436(30)	**436.0**
LA16	1169(30)	**1169.0**	1169(30)	**1169.0**	1169(29)	1171.1
LA17	899(29)	**900.2**	899(6)	1156.6	899(26)	938.2
LA18	929(30)	**929.0**	929(20)	931.3	929(30)	**929.0**
LA19	948(9)	**1005.9**	948(7)	1066.0	948(1)	1058.0
LA20	805(4)	842.2	805(1)	849.0	805(6)	**840.0**
LA21	463(30)	463.0	463(30)	463.0	463(30)	463.0
LA22	1064(3)	**1084.1**	1084(5)	1153.5	1064(4)	1147.9
LA23	835(3)	870.4	835(11)	**860.8**	835(1)	873.2
LA24	835(30)	**835.0**	835(21)	881.4	835(30)	**835.0**
MT10	1363(29)	**1371.4**	1363(5)	1720.4	1363(26)	1389.3
ORB1	2568(18)	2608.4	2568(1)	2987.6	2568(15)	**2596.6**
ORB2	1408(2)	1431.8	1408(5)	**1429.7**	1408(1)	1433.1
ORB3	2111(18)	2143.0	2111(5)	2616.1	2111(21)	**2130.4**
ORB4	1623(20)	1636.5	1623(18)	1672.3	1623(28)	**1624.5**
ORB5	1593(12)	**1642.1**	1667(6)	1755.8	1593(8)	1647.0
ORB6	1790(24)	1802.8	1790(19)	1976.5	1790(29)	**1790.1**
ORB7	590(28)	591.6	590(14)	599.3	590(28)	**591.2**
ORB8	2429(17)	2449.9	2453(1)	2936.4	2429(13)	**2449.7**
ORB9	1316(30)	**1316.0**	1316(18)	1368.4	1316(30)	**1316.0**
ORB10	1679(25)	1699.1	1679(4)	1896.7	1679(26)	**1696.9**

Hemos elegido los parámetros de los diferentes métodos para que los tiempos de ejecución sean lo más parecidos posible. En cuanto a la búsqueda tabú y el genético combinado con escalada, hemos realizado un estudio experimental para elegir los parámetros que mejores resultados ofrecen. La búsqueda tabú se ha realizado con la estructura N^S utilizando todos los caminos críticos, una lista de soluciones elite de tamaño 10, y un máximo de iteraciones de 120000. El genético combinado con escalada de máximo gradiente utiliza la estructura N^S y elige en cada cromosoma de forma aleatoria si va a utilizar uno o todos los caminos críticos. Por otra parte el tamaño de la población es de 260 individuos, y el número de generaciones

de 100. Omitimos los detalles del estudio realizado para elegir estos parámetros por no considerarlo de gran interés, debido a que el método que elegiremos finalmente es el genético combinado con búsqueda tabú.

En la tabla hemos marcado en negrita los valores de weighted tardiness medios que superan a los obtenidos por las demás configuraciones. Se puede apreciar que la búsqueda tabú por separado es el método que ofrece peores resultados, mientras que los otros dos métodos son ambos muy eficientes y consiguen medias de weighted tardiness similares. Debido a que los resultados son tan parecidos, para decantarnos por un método hemos calculado los errores porcentuales de las medias obtenidas, de la misma forma que hicimos en la sección 10.3.2. El resultado es que el error porcentual medio de AG+TS es 1.36 %, el de TS es 8.92 % y el de AG+BL es 2.01 %, por lo tanto AG+TS resulta ser el mejor algoritmo teniendo en cuenta las medias de weighted tardiness. Además, el número de ejecuciones en las que AG+TS encuentra la mejor solución conocida es de 446, mientras que AG+BL lo consigue en 442 ejecuciones. Por último, AG+TS parece ser un método más robusto, ya que en el caso de AG+BL hay tres instancias en las que sólo ha llegado en 1 de las 30 ejecuciones a la mejor solución conocida, mientras que en la combinación con la búsqueda tabú siempre llega en dos o más ejecuciones. Por todos estos motivos, aunque los dos métodos ofrecen resultados excelentes, hemos elegido la combinación de algoritmo genético y búsqueda tabú.

10.3.4. Comparación con el estado del arte. Primer conjunto de instancias

Proponemos para realizar este estudio experimental el método que mejores resultados obtuvo en los experimentos previos, el cual resumimos a continuación. Se trata de una combinación del algoritmo genético descrito en el capítulo 5 con la búsqueda tabú descrita en la sección 6.4. No consideraremos la gestión de la lista de soluciones elite para la búsqueda tabú, ya que al combinarla con el genético utilizaremos un número máximo de iteraciones bastante reducido para que el tiempo de ejecución no sea excesivo. A cada cromosoma generado por el algoritmo genético se le aplicará el constructor de planificacones SSGS, descrito en la sección 3.3.2. Por otra parte utilizamos la estructura de vecindad N_1^S y el método de estimación descritos en el capítulo 7. Además decidimos aleatoriamente en cada cromososa si utilizar únicamente el camino crítico más importante o todos los caminos críticos, ya que en la sección anterior comprobamos que es la mejor opción con el tiempo de ejecución disponible. Denominaremos a partir de este momento $AG+TS-N_1^S$ al algoritmo

propuesto para el resto de los estudios experimentales de este capítulo.

Como ya hemos comentado, el algoritmo propuesto por Essafi et al. esta implementado en C++ y los experimentos se ejecutan en un PC con un procesador de 2.8 GHz y 512 MB de RAM, dejando a cada ejecución un tiempo máximo de 18 segundos. El método de búsqueda local de Mati et al. se ejecuta en un Pentium con un procesador de 2.6 GHz, y también deciden dar a cada ejecución un tiempo máximo de 18 segundos. Por otra parte el algoritmo de Kreipl se ejecuta en un Pentium 233 MHz y dan un tiempo máximo de 200 segundos a cada ejecución. Por último, $AG + TS - N_1^S$ se ejecuta en Windows XP en un Intel Core 2 Duo a 2.66GHz con 2Gb de RAM. Debido a que es un ordenador similar a los utilizados por Essafi et al. y por Mati et al., decidimos que el tiempo de ejecución de $AG + TS - N_1^S$ se sitúe también alrededor de los 18 segundos, y para ello los parámetros elegidos son: 58 indivíduos en la población, 70 generaciones, y 50 iteraciones máximas sin mejora en la búsqueda tabú.

En las tablas 10.5, 10.6 y 10.7 se muestran los experimentos utilizando $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$ respectivamente. En todas ellas la primera columna indica el nombre de la instancia. La segunda columna (SP) muestra la solución obtenida con el método de ramificación y poda propuesto por Singer y Pinedo en [132]. La tercera columna (LSRW) muestra los resultados del algoritmo de búsqueda local de [79], se indica el resultado medio y entre paréntesis el número de veces que obtuvo la mejor solución conocida en las 5 ejecuciones realizadas. Las dos siguientes columnas (GLS) muestran los resultados del algoritmo genético híbrido de [49], en la primera se indica el mejor resultado y entre paréntesis el número de veces que se obtuvo la mejor solución conocida, y en la segunda el resultado medio. Las dos siguientes columnas (MDL) muestran los resultados obtenidos por el método de búsqueda local propuesto por Mati et al. en [86]. Las siguientes columnas corresponden a los resultados de nuestro método $AG+TS-N_1^S$, en los que indicamos el mejor resultado obtenido, entre paréntesis el número de ejecuciones en las que se ha llegado a la mejor solución conocida, el resultado medio y el tiempo medio de cada ejecución. En las tablas un "*" indica que el correspondiente valor es igual que la solución obtenida por el algoritmo SP, y un "+" indica que dicha solución se supera. En todos los casos la mejor solución conocida para cada instancia será el mejor valor encontrado por nuestro método.

Observaremos que en varias instancias con $f = 1{,}3$ y en una instancia con $f = 1{,}6$ ocurre que, tanto nuestro algoritmo como GLS y MDL obtienen una solución mejor que las encontradas por el procedimiento de ramificación y poda SP propuesto por Singer y

Pinedo. Sospechamos que las soluciones proporcionadas en [132] no son óptimas porque detuvieron su algoritmo de ramificación y poda antes de explorar todo el árbol de búsqueda, y eso explicaría por qué existen soluciones con una función objetivo más pequeña en algunas instancias. O quizás sea algún tipo de problema de redondeo que tuvieron al generar los due dates, y eso haría que el óptimo les resultara ligeramente distinto. De todas formas formas es un hecho que ya ha sido observado por distintos investigadores en otros trabajos.

En todos los casos, tanto en GLS como en MDL como en nuestro algoritmo, se realizan 10 ejecuciones para cada instancia. Debemos tener en cuenta también que Kreipl ejecuta únicamente 5 pruebas de su método, y por lo tanto está en desventaja con los restantes métodos en cuanto a ser capaz de alcanzar la mejor solución conocida en alguna de las ejecuciones realizadas.

La tabla 10.5 muestra los resultados de los experimentos utilizando $f = 1{,}3$. Observamos que nuestro método es el único capaz de obtener la mejor solución conocida en alguna ejecución en todas y cada una de las instancias. LSRW falla en 9 instancias (aunque recordemos que esta en desventaja debido a lanzar únicamente 5 ejecuciones), mientras que GLS falla en 2 instancias y MDL también en 2 instancias. En la instancia ORB8 nuestro método es el único capaz de encontrar la mejor solución conocida, y además lo consigue en 4 de las 10 ejecuciones. Globalmente vemos que es un método muy robusto, ya que siempre consigue llegar a la mejor solución conocida en tres o más ejecuciones, exceptuando la instancia LA23 en la que sólo lo ha conseguido en una ejecución.

En [146], Van Hentenryck y Michel ofrecen resultados sobre las instancias con $f = 1{,}3$. No los hemos mostrado en la tabla correspondiente por no considerar los resultados mejores que los obtenidos por GLS o MDL. Sin embargo realizaremos una concisa comparación con ellos. Lanzan 50 ejecuciones de su método para cada instancia, y consiguen alguna vez la mejor solución conocida en todas ellas, excepto en una (ORB7). Del total de las 1100 ejecuciones que realizan, Van Hentenryck y Michel obtienen la mejor solución conocida en 617 ejecuciones (56.1 %). Nuestro método la obtiene en 152 de las 220 ejecuciones (69.1 %), GLS la obtiene en 143 de las 220 ejecuciones (65.0 %), MDL en 133 de las 220 ejecuciones (60.5 %) y LSRW en 53 de las 110 ejecuciones (48.2 %).

Para realizar este análisis, en el caso de los métodos propuestos por Kreipl (LSRW) y por Van Hentenryck y Michel, cuando en sus trabajos indican que obtienen los resultados óptimos asumimos que son los mismos encontrados por nuestro algoritmo, GLS y MDL en los casos en los que dicho resultado es menor que el ofrecido por SP. Si ese no fuera el caso,

Tabla 10.5: Experimentos weighted tardiness: Comparación con el estado del arte en el JSP. Instancias 10×10. $f = 1{,}3$

Inst.	SP	$LSRW$	GLS		MDL		$AG + TS - N_1^S$		
			Mejor	Media	Mejor	Media	Mejor	Media	T(s)
ABZ5	1405	1451(0)	1403(10)+	1403+	1403(2)+	1414	1403(7)+	1411.7	18.3
ABZ6	436	*	*(10)	*	*(10)	*	*(10)	*	15.6
LA16	1170	*	1169(9)+	1175	1169(10)+	1169+	1169(10)+	1169.0+	17.5
LA17	900	*	899(10)+	899+	899(10)+	899+	899(10)+	899.0+	15.9
LA18	929	*	*(6)	933	*(6)	934	*(10)	*	16.7
LA19	948	951(3)	*(8)	949	*(10)	*	*(4)	997.5	19.2
LA20	809	*	805(10)+	805+	805(10)+	805+	805(3)+	833.7	16.8
LA21	464	*	463(10)+	463+	463(10)+	463+	463(10)+	463.0+	16.3
LA22	1068	1086(0)	1064(1)+	1087	1064(4)+	1077	1064(3)+	1078.7	17.7
LA23	837	875(0)	835(2)+	865	835(2)+	865	835(1)+	870.0	17.6
LA24	835	*	*(10)	*	*(10)	*	*(10)	*	16.8
MT10	1368	*	1363(9)+	1372	1363(10)+	1363+	1363(9)+	1382.7	18.7
ORB1	2568	2616(2)	2570(0)	2651	*(3)	2639	*(8)	2578.0	20.8
ORB2	1412	1434(0)	1408(2)+	1444	1408(3)+	1426	1408(3)+	1426.2	18.2
ORB3	2113	2204(0)	2111(4)+	2170	2111(1)+	2158	2111(6)+	2159.8	20.0
ORB4	1623	1674(1)	*(7)	1643	*(2)	1690	*(6)	1631.8	18.7
ORB5	1593	1662(0)	*(1)	1659	1738(0)	1775	*(7)	1615.2	19.0
ORB6	1792	1802(4)	1790(10)+	1790+	1790(9)+	1793	1790(5)+	1854.0	19.6
ORB7	590	618(0)	*(9)	592	*(10)	*	*(10)	*	19.8
ORB8	2429	2554(0)	2439(0)	2522	2461(0)	2523	*(4)	2476.7	20.9
ORB9	1316	1334(3)	*(10)	*	*(10)	*	*(10)	*	17.1
ORB10	1679	1775(0)	*(5)	1718	*(1)	1774	*(6)	1730.6	19.5

*: solución igual que SP, +: solución mejor que SP

esos dos métodos serían peores que lo indicado en este análisis.

La tabla 10.6 muestra los experimentos utilizando $f = 1{,}5$. Observamos que únicamente nuestro método y GLS son capaces de obtener la mejor solución conocida en alguna ejecución en todas y cada una de las instancias. MDL falla en 3 instancias y LSRW en 3 instancias. Si comparamos nuestro método con los demás, $AG + TS - N_1^S$ alcanza la mejor solución conocida en 179 de las 220 ejecuciones (81.4 %), mientras que GLS la alcanza en 182 de las 220 ejecuciones (82.7 %), MDL en 164 de las 220 ejecuciones (74.5 %), y LSRW en 73 de las 110 ejecuciones (66.4 %).

La tabla 10.7 muestra los experimentos utilizando $f = 1{,}6$. Observamos que, de nuevo, nuestro método es el único que es capaz de conseguir alguna vez la mejor solución conocida

Tabla 10.6: Experimentos weighted tardiness: Comparación con el estado del arte en el JSP. Instancias 10×10. $f = 1{,}5$

Inst.	SP	$LSRW$	GLS		MDL		$AG+TS-N_1^S$		
			Mejor	Media	Mejor	Media	Mejor	Media	T(s)
ABZ5	69	70(3)	*(10)	*	*(10)	*	*(10)	*	16.5
ABZ6	0	*	*(10)	*	*(10)	*	*(10)	*	6.1
LA16	166	*	*(10)	*	*(10)	*	*(9)	166.3	16.5
LA17	260	*	*(10)	*	*(10)	*	*(10)	*	14.3
LA18	34	*	*(10)	*	*(10)	*	*(10)	*	15.2
LA19	21	*	*(10)	*	*(10)	*	*(10)	*	16.8
LA20	0	*	*(10)	*	*(10)	*	*(7)	0.3	13.5
LA21	0	*	*(10)	*	*(10)	*	*(10)	*	12.1
LA22	196	*	*(10)	*	*(10)	*	*(10)	*	16.0
LA23	2	*	*(10)	*	*(10)	*	*(10)	*	12.3
LA24	82	90(1)	*(3)	86	*(2)	86	*(1)	88.0	15.0
MT10	394	414(1)	*(10)	*	*(10)	*	*(10)	*	17.0
ORB1	1098	1143(0)	*(6)	1159	1196(0)	1247	*(10)	*	19.8
ORB2	292	*	*(10)	*	*(10)	*	*(10)	*	16.5
ORB3	918	965(1)	*(4)	943	952(0)	961	*(4)	938.7	20.7
ORB4	358	*	*(8)	394	*(4)	435	*(10)	*	16.6
ORB5	405	455(1)	*(10)	*	*(8)	415	*(7)	428.1	16.8
ORB6	426	*	*(5)	440	*(5)	437	*(8)	430.0	18.4
ORB7	50	119(0)	*(8)	55	*(10)	*	*(10)	*	17.4
ORB8	1023	1138(0)	*(7)	1059	*(6)	1036	*(2)	1032.6	18.9
ORB9	297	*	*(7)	311	*(9)	299	*(9)	301.7	17.7
ORB10	346	408(1)	*(4)	400	424(0)	436	*(2)	430.0	18.6

*: solución igual que SP, +: solución mejor que SP

en todas y cada una de las instancias. GLS no consigue llegar a dicha solución en 4 de las instancias, y MDL no consigue llegar a ella en 2 de las instancias. LSRW no consigue llegar en 2 de las instancias, aunque recordemos que LSRW está en desventaja en este aspecto, debido a que sólo lanzan 5 ejecuciones de su algoritmo. De hecho nuestro algoritmo es el único método capaz de llegar a la mejor solución conocida en las instancias ORB1 y ORB10, consiguiéndolo además en dos ejecuciones y en cinco ejecuciones, respectivamente. Si comparamos nuestro método con los demás, alcanzamos la mejor solución conocida en 186 de las 220 ejecuciones (84.5 %), mientras que GLS la alcanza en 157 de las 220 ejecuciones (71.4 %), MDL en 161 de las 220 ejecuciones (73.2 %), y LSRW en 88 de las 110 ejecuciones (80.0 %). En cuanto a los resultados medios, nuestro algoritmo resulta ser igual o mejor, en

Tabla 10.7: Experimentos weighted tardiness: Comparación con el estado del arte en el JSP. Instancias 10×10. $f = 1{,}6$

Inst.	SP	$LSRW$	GLS		MDL		$AG+TS-N_1^S$		
			Mejor	Media	Mejor	Media	Mejor	Media	T(s)
ABZ5	0	*	*(10)	*	*(10)	*	*(10)	*	8.2
ABZ6	0	*	*(10)	*	*(10)	*	*(10)	*	2.7
LA16	0	*	*(10)	*	*(10)	*	*(10)	*	12.7
LA17	65	*	*(10)	*	*(10)	*	*(10)	*	14.1
LA18	0	*	*(10)	*	*(10)	*	*(10)	*	10.1
LA19	0	*	*(10)	*	*(10)	*	*(10)	*	8.6
LA20	0	*	*(10)	*	*(10)	*	*(10)	*	5.1
LA21	0	*	*(10)	*	*(10)	*	*(10)	*	4.5
LA22	0	*	*(10)	*	*(10)	*	*(10)	*	12.1
LA23	0	*	*(10)	*	*(10)	*	*(10)	*	4.7
LA24	0	*	*(10)	*	*(10)	*	*(10)	*	9.2
MT10	141	144(3)	*(1)	162	*(1)	152	*(6)	144.5	16.5
ORB1	566	624(0)	604(0)	688	604(0)	653	*(2)	591.5	20.2
ORB2	44	*	*(10)	*	*(10)	*	*(10)	*	14.2
ORB3	422	441(2)	*(1)	514	*(4)	463	*(7)	433.7	19.9
ORB4	66	*	*(8)	78	*(8)	68	*(10)	*	15.8
ORB5	163	174(2)	181(0)	181	*(3)	176	*(3)	175.6	14.6
ORB6	31	*	28(10)+	28+	28(10)+	28+	28(10)+	28.0+	17.2
ORB7	0	*	*(10)	*	*(10)	*	*(10)	*	13.0
ORB8	621	658(1)	646(0)	669	*(1)	643	*(3)	639.0	18.6
ORB9	66	*	*(7)	83	*(4)	80	*(10)	*	15.6
ORB10	76	97(0)	84(0)	142	78(0)	117	*(5)	82.0	18.9

*: solución igual que SP, +: solución mejor que SP

todos los casos, que GLS y MDL. Si comparamos las medias obtenidas por nuestro algoritmo con las obtenidas por LSRW, resultan mejores en 4 instancias, peores en 2 instancias, e idénticas en las restantes 16 instancias.

Globalmente, $AG+TS-N_1^S$ es el único método que ha sido capaz de obtener la mejor solución conocida en alguna ejecución en las 66 instancias de este benchmark. Además resulta ser un método muy robusto, ya que dicha solución se consigue casi siempre en varias ejecuciones; únicamente en 2 de las 66 instancias nuestro método la consigue en una única ejecución de las 10 realizadas. Globalmente, $AG+TS-N_1^S$ ha obtenido la mejor solución conocida en 517 del total de 660 ejecuciones (78.3 %), mientras que GLS la ha obtenido en 443 del total de 660 ejecuciones (67.1 %), MDL en 458 del total de 660 ejecuciones (69.4 %),

y LSRW en 214 del total de 330 ejecuciones (64.8 %).

Además también debemos observar que el tiempo de ejecución utilizado por $AG+TS-N_1^S$ es generalmente menor de los 18 segundos que utilizan tanto GLS como MDL. En concreto, el tiempo medio utilizado en las instancias con $f = 1{,}3$ es de 18.2 segundos por prueba, con $f = 1{,}5$ de 16.0 segundos, y con $f = 1{,}6$ de 12.6 segundos. Sólo en 5 de las 66 instancias el tiempo medio por prueba supera los 20 segundos, y es siempre inferior a 21 segundos.

Por todos estos motivos, los resultados en este primer conjunto de instancias muestran que nuestro algoritmo resulta una alternativa mejor, o como mínimo muy competitiva, con los demás de métodos del estado del arte.

10.3.5. Comparación con el estado del arte. Segundo conjunto de instancias

Las tablas 10.8, 10.9 y 10.10 muestran los resultados en el segundo conjunto de instancias considerado, para $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$ respectivamente. Indicamos los resultados obtenidos por nuestro algoritmo $AG+TS-N_1^S$ junto con los del algoritmo GLS presentados en [49]. En dicho trabajo lanzan 10 ejecuciones de su algoritmo genético híbrido, e indican que cada una de las ejecuciones se finaliza tras 200 generaciones, pero no se dan más detalles del tiempo de ejecución utilizado en estos experimentos, por lo que no podemos hacer una comparación del todo precisa. Además, tampoco detallan que hayan utilizado ningún límite de tiempo, por lo que suponemos que utilizando la misma configuración, el tiempo de ejecución empleado en las instancias más grandes es mucho mayor que el empleado en las instancias más reducidas. Por este motivo, nosotros en estos experimentos también elegimos la misma configuración para todas las 40 instancias, sin tener en cuenta su tamaño. Los parámetros elegidos para $AG+TS-N_1^S$ son 100 individuos en la población, 200 generaciones, y se aplica la búsqueda tabú a cada cromosoma hasta que transcurran 50 iteraciones consecutivas sin encontrar mejora. Los parámetros utilizados son mayores que los de la sección anterior debido a que las instancias son por lo general más grandes y previsiblemente más difíciles.

En la tabla 10.8 se muestran los resultados con $f = 1{,}3$. En cuanto a los mejores weighted tardiness, nuestro método obtiene mejores resultados que GLS en 7 instancias, iguales en 19 instancias y peores en 14 instancias. En cuanto a los weighted tardiness medios, nuestro método obtiene mejores resultados que GLS en 18 instancias, iguales en 8 instancias, y peores en 14 instancias. Debemos destacar que $AG+TS-N_1^S$ obtiene los peores resultados, comparativamente, en las instancias de tamaño 30×10, ya que en todos los casos obtiene

peores resultados, tanto en el mejor como en la media. Por el contrario, observamos que en la totalidad de las instancias de tamaño 20×5 y 15×15 nuestro algoritmo ha obtenido una media mejor que la de GLS.

Las altas desviaciones estándar parecen anunciar que la media aún oscila mucho de una ejecución a otra, lo que puede significar que nuestro algoritmo aún no ha llegado a converger adecuadamente, y por tanto tiene margen para mejorar estos resultados. Además, las desviaciones son mayores precisamente en las instancias LA31 a LA35, que son en las que peor se comporta nuestro algoritmo respecto a GLS. Sin embargo la comparación no puede ser del todo precisa debido a que no conocemos el tiempo de ejecución empleado por Essafi et al. en estos experimentos y es posible tanto que haya sido menor como que haya sido mucho mayor, particularmente en las instancias 30×10.

Sobre la tabla 10.9 y los resultados con $f = 1{,}5$, nuestro método obtiene un mejor weighted tardiness que GLS en 9 instancias, igual en 18 instancias y peor en 13 instancias. Sobre los weighted tardiness medios, nuestro método obtiene uno mejor que GLS en 14 instancias, igual en 9 instancias, y peor en 17 instancias. De nuevo podemos observar que nuestro método obtiene los peores resultados en las instancias de tamaño 30×10.

Por último, la tabla 10.10 presenta los resultados con $f = 1{,}6$. Observando los mejores weighted tardiness alcanzados, vemos que nuestro método alcanza uno mejor que GLS en 5 instancias, igual en 20 instancias y peor en 15 instancias. En cuanto a los weighted tardiness medios, $AG+TS-N_1^S$ alcanza uno mejor que GLS en 19 instancias, igual en 14 instancias, y peor en 7 instancias. Nuevamente nuestro método obtiene los peores resultados en las instancias de tamaño 30×10, mientras que en casi todos los demás tamaños de instancia nuestro método obtiene casi siempre una media igual o mejor.

Tabla 10.8: Experimentos weighted tardiness: Comparación con el estado del arte en el JSP. Instancias LA. $f = 1{,}3$

Inst.	Tamaño	*GLS*		$AG + TS - N_1^S$			
		Mejor	Media	Mejor	Media	Desv	T(s)
LA01	10×5	2299	2299	2299(10)	2299.0	0.0	46
LA02	10×5	1762	1762	1762(10)	1762.0	0.0	44
LA03	10×5	1951	1951	1951(10)	1951.0	0.0	45
LA04	10×5	1917	1917	1917(10)	1917.0	0.0	45
LA05	10×5	1878	1878	1878(10)	1878.0	0.0	45
LA06	15×5	5810	**5827**	5810(3)	5964.4	109.7	113
LA07	15×5	5765	**5801**	5765(5)	5808.0	45.3	116
LA08	15×5	5475	**5482**	5475(2)	5533.4	30.8	121
LA09	15×5	5608	5648	5608(10)	**5608.0**	0.0	111
LA10	15×5	6618	6621	6618(10)	**6618.0**	0.0	98
LA11	20×5	12039	12341	12009(4)	**12200.3**	177.9	246
LA12	20×5	10542	10683	10542(5)	**10606.8**	92.7	243
LA13	20×5	11557	11889	11502(1)	**11608.4**	83.6	245
LA14	20×5	13107	13225	13128(1)	**13174.9**	47.8	226
LA15	20×5	12330	12428	12278(2)	**12363.6**	55.3	236
LA16	10×10	1169	1169	1169(10)	1169.0	0.0	77
LA17	10×10	899	899	899(10)	899.0	0.0	71
LA18	10×10	929	929	929(10)	929.0	0.0	78
LA19	10×10	948	**948**	948(9)	948.7	2.2	86
LA20	10×10	805	**805**	805(2)	837.8	17.3	77
LA21	15×10	3560	**3771**	3560(1)	3856.6	141.0	206
LA22	15×10	4227	4471	4291(8)	**4316.7**	54.3	215
LA23	15×10	3777	**3955**	3777(2)	3957.4	128.7	208
LA24	15×10	3553	3831	3539(1)	**3696.5**	57.5	203
LA25	15×10	3313	3569	3313(4)	**3410.0**	95.4	214
LA26	20×10	9093	9748	8820(1)	**9416.4**	354.9	451
LA27	20×10	9127	**9860**	10050(1)	10315.8	216.8	496
LA28	20×10	9263	9757	9244(1)	**9577.6**	228.8	472
LA29	20×10	8744	9397	8939(1)	**9354.3**	283.0	489
LA30	20×10	8626	**8968**	8663(1)	9059.9	208.0	435
LA31	30×10	27671	**28999**	30020(1)	30935.7	615.3	1914
LA32	30×10	30301	**32004**	32619(1)	34228.3	944.4	1931
LA33	30×10	25902	**27216**	27413(1)	28248.7	744.2	1856
LA34	30×10	27901	**29131**	30331(1)	31138.6	611.5	1943
LA35	30×10	29652	**30979**	30804(1)	31750.3	527.8	1881
LA36	15×15	2957	3187	3014(1)	**3131.7**	85.3	312
LA37	15×15	2290	2880	2517(3)	**2580.3**	80.8	285
LA38	15×15	2159	2287	2087(1)	**2275.7**	146.1	310
LA39	15×15	1676	1861	1718(1)	**1820.4**	98.0	309
LA40	15×15	2078	2305	2155(1)	**2206.8**	27.1	329

Tabla 10.9: Experimentos weighted tardiness: Comparación con el estado del arte en el JSP. Instancias LA. $f = 1{,}5$

Inst.	Tamaño	GLS		$AG + TS - N_1^S$			
		Mejor	Media	Mejor	Media	Desv	T(s)
LA01	10×5	1610	1610	1610(10)	1610.0	0.0	40
LA02	10×5	1028	1028	1028(10)	1028.0	0.0	43
LA03	10×5	1280	1280	1280(10)	1280.0	0.0	47
LA04	10×5	1277	1277	1277(10)	1277.0	0.0	43
LA05	10×5	1205	1205	1205(10)	1205.0	0.0	42
LA06	15×5	4592	**4658**	4597(1)	4685.1	31.7	115
LA07	15×5	4470	4548	4470(10)	**4470.0**	0.0	114
LA08	15×5	4034	4094	4034(8)	**4079.2**	95.3	118
LA09	15×5	4419	**4421**	4419(8)	4423.7	10.2	118
LA10	15×5	5148	**5148**	5148(7)	5161.8	22.2	100
LA11	20×5	10170	10332	10054(1)	**10187.1**	68.4	246
LA12	20×5	8982	**9084**	8981(1)	9104.9	51.2	250
LA13	20×5	9626	9846	9630(1)	**9677.5**	50.0	234
LA14	20×5	11286	**11382**	11276(1)	11412.1	70.2	221
LA15	20×5	10343	10455	10154(1)	**10376.5**	98.7	239
LA16	10×10	166	166	166(10)	166.0	0.0	75
LA17	10×10	260	260	260(10)	260.0	0.0	66
LA18	10×10	34	34	34(10)	34.0	0.0	68
LA19	10×10	21	21	21(10)	21.0	0.0	78
LA20	10×10	0	**0**	0(8)	0.2	0.4	63
LA21	15×10	1570	**1692**	1706(1)	1760.2	33.6	201
LA22	15×10	2015	2273	2080(2)	**2264.0**	128.9	216
LA23	15×10	1510	1683	1510(10)	**1510.0**	0.0	189
LA24	15×10	1570	**1618**	1570(2)	1621.2	66.2	199
LA25	15×10	1430	1497	1430(9)	**1439.1**	28.8	193
LA26	20×10	5369	**6106**	5973(1)	6324.0	221.2	438
LA27	20×10	5793	**6142**	5678(1)	6303.2	339.9	490
LA28	20×10	5744	6254	5925(1)	**6095.7**	146.9	467
LA29	20×10	5959	6392	5802(1)	**6026.5**	246.0	475
LA30	20×10	5259	**5496**	5094(1)	5539.2	293.5	434
LA31	30×10	22545	**23417**	23498(1)	24720.0	781.5	1872
LA32	30×10	24467	**25766**	26288(1)	27363.8	755.6	1944
LA33	30×10	20493	**21767**	21290(1)	22067.2	656.7	1920
LA34	30×10	22572	**23341**	22710(1)	23973.8	1040.7	1884
LA35	30×10	24325	**24654**	24621(1)	25841.2	587.1	1899
LA36	15×15	590	866	581(1)	**589.0**	2.8	282
LA37	15×15	413	589	414(9)	**415.4**	4.4	250
LA38	15×15	401	438	393(1)	**416.4**	23.6	271
LA39	15×15	0	9	0(3)	**7.1**	8.9	236
LA40	15×15	51	**79**	74(3)	83.6	14.2	268

Tabla 10.10: Experimentos weighted tardiness: Comparación con el estado del arte en el JSP. Instancias LA. $f = 1{,}6$

Inst.	Tamaño	*GLS*		$AG + TS - N_1^S$			
		Mejor	Media	Mejor	Media	Desv	T(s)
LA01	10×5	1230	1230	1230(10)	1230.0	0.0	46
LA02	10×5	695	695	695(10)	695.0	0.0	42
LA03	10×5	1024	1024	1024(10)	1024.0	0.0	46
LA04	10×5	1029	1029	1029(10)	1029.0	0.0	40
LA05	10×5	877	877	877(10)	877.0	0.0	41
LA06	15×5	4098	4130	4116(2)	**4121.6**	6.5	119
LA07	15×5	3843	3988	3843(10)	**3843.0**	0.0	105
LA08	15×5	3400	**3400**	3400(9)	3420.6	65.1	114
LA09	15×5	3811	3835	3811(9)	**3814.5**	11.1	118
LA10	15×5	4503	4533	4503(9)	**4512.8**	31.0	105
LA11	20×5	9232	9399	9232(1)	**9300.6**	48.8	249
LA12	20×5	8255	8302	8178(2)	**8258.2**	66.0	241
LA13	20×5	8767	8916	8761(1)	**8845.4**	109.9	232
LA14	20×5	10496	10594	10519(1)	**10564.1**	25.9	221
LA15	20×5	9284	9392	9299(1)	**9365.3**	49.6	237
LA16	10×10	0	0	0(10)	0.0	0.0	47
LA17	10×10	65	65	65(10)	65.0	0.0	65
LA18	10×10	0	0	0(10)	0.0	0.0	40
LA19	10×10	0	0	0(10)	0.0	0.0	29
LA20	10×10	0	0	0(10)	0.0	0.0	20
LA21	15×10	868	949	905(5)	**914.8**	10.6	196
LA22	15×10	1242	1450	1364(1)	**1403.8**	19.1	219
LA23	15×10	936	977	941(9)	**945.6**	14.6	178
LA24	15×10	752	773	693(1)	**748.9**	20.0	182
LA25	15×10	874	922	874(1)	**892.5**	8.3	185
LA26	20×10	4159	5125	4590(1)	**4889.6**	238.0	434
LA27	20×10	4307	4590	4049(1)	**4396.1**	254.2	481
LA28	20×10	4248	4594	4293(1)	**4545.3**	234.5	452
LA29	20×10	4392	4706	4362(1)	**4569.3**	97.9	440
LA30	20×10	3631	**4131**	3941(1)	4211.1	206.7	417
LA31	30×10	20006	**21141**	20838(1)	21827.0	644.6	1912
LA32	30×10	21722	**22918**	21979(1)	24038.1	1093.7	1918
LA33	30×10	17771	**18801**	18408(1)	19417.0	720.3	1833
LA34	30×10	19693	**20548**	20691(1)	21589.1	594.3	1889
LA35	30×10	20601	**21991**	22093(1)	22988.4	733.5	1880
LA36	15×15	116	192	118(3)	**150.6**	22.5	241
LA37	15×15	0	0	0(10)	0.0	0.0	181
LA38	15×15	0	0	0(10)	0.0	0.0	209
LA39	15×15	0	0	0(10)	0.0	0.0	140
LA40	15×15	0	0	0(10)	0.0	0.0	177

Globalmente, si manejamos los datos del total de 120 instancias, $AG+TS-N_1^S$ obtiene una mejor media de weighted tardiness en 51 instancias (42.5 %), la misma media en 31 instancias (25.8 %) y una peor media en 38 instancias (31.7 %). Estos resultados se consiguen a pesar de que en el total de 15 instancias de tamaño 30×10 hemos obtenido una media peor, ya que para el resto de tamaños nuestro método obtiene, por lo general, medias iguales o mejores que GLS. De todas formas, ya comentamos que las altas desviaciones estándar mostradas pueden indicar que nuestro algoritmo todavía podría mejorar más los resultados disponiendo de un tiempo de ejecución mayor.

En conclusión, los resultados en este segundo conjunto de instancias muestran que no se puede decir que nuestro algoritmo sea más eficiente que el propuesto por Essafi et al., ya que al no conocer el tiempo de ejecución empleado por su algoritmo, la comparación no puede ser del todo precisa.

10.4. Minimización del weighted tardiness en el SDST-JSP

Como ya hemos comentado en la sección 10.1, no nos ha sido posible compararnos con el único trabajo que conocemos referido a la minimización del weighted tardiness en el SDST-JSP. Debido a este problema, decidimos modelar el problema con el ILOG CPLEX CP Optimizer (CP), y comparar nuestros resultados con los obtenidos por dicho método. Los detalles del método CP se pueden consultar en la sección 4.4.3. Hemos utilizado en concreto las instancias del conjunto BT y el conjunto de instancias propuesto por Vela et al. en [151]. Los detalles de estas instancias se pueden consultar en las secciones 8.2.1 y 8.2.3. Generamos los due dates y pesos para estas instancias con el método descrito en la sección 10.2, utilizando de nuevo los valores $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$, al igual que en el JSP clásico.

La configuración utilizada por $AG+TS-N_1^S$ para estos experimentos es la misma que la utilizada en la sección anterior con el conjunto de instancias LA01-40, es decir, 100 individuos en la población, 200 generaciones, y aplicamos la búsqueda tabú hasta que transcurran 50 iteraciones consecutivas sin mejora. En los experimentos con CP, lo hemos ejecutado por un tiempo aproximadamente 20 % superior al tiempo utilizado por $AG+TS-N_1^S$. Para los dos métodos realizamos 30 ejecuciones para cada una de las instancias.

Las tablas 10.11, 10.12 y 10.13 muestran los resultados de las series de experimentos

realizadas con $f = 1{,}3$, $f = 1{,}5$ y $f = 1{,}6$ respectivamente. Para cada uno de los dos métodos indicamos el mejor weighted tardiness alcanzado y el número de veces que se alcanza, el weighted tardiness medio, la desviación estándar y el tiempo medio consumido por una única ejecución.

Observamos que en casi todos los casos el weighted tardiness medio obtenido por $AG + TS - N_1^S$ es mejor que el obtenido por CP, siendo además menor el tiempo de ejecución. Las dos únicas excepciones son la instancia ABZ8sdst con $f = 1{,}3$ y la instancia ABZ9sdst con $f = 1{,}5$, para las que CP obtiene un valor medio ligeramente mejor que nosotros. En término medio CP consigue un weighted tardiness medio un 13.6 % superior al de $AG + TS - N_1^S$, aunque las diferencias aumentan con el valor del parámetro f. Para las instancias con $f = 1{,}3$ el weighted tardiness medio de CP es un 10.3 % superior, mientras que en las instancias con $f = 1{,}5$ es un 13.8 % superior, y en las instancias con $f = 1{,}6$ es un 16.6 % superior.

En cuanto al tamaño de la instancia, las menores diferencias entre los dos métodos se aprecian en las instancias ABZ7sdst, ABZ8sdst y ABZ9sdst, es decir las instancias de mayor tamaño de entre todas las consideradas, en las que CP obtiene un weighted tardiness medio un 2.3 % superior. Las mayores diferencias aparecen en las instancias LA38 y LA40, ya que en esas dos instancias CP obtiene en media un weighted tardiness un 38.8 % superior, siendo la diferencia especialmente acusada en el caso con $f = 1{,}6$.

En cuanto al mejor weighted tardiness alcanzado por $AG+TS-N_1^S$ en las 30 ejecuciones, es mejor en todas las instancias al alcanzado por CP en sus 30 ejecuciones. Por todos estos motivos, concluímos que los resultados de nuestro algoritmo en esta serie de experimentos superan a los obtenidos por CP, aunque estos últimos son competitivos, especialmente en las instancias de mayor tamaño.

Tabla 10.11: Experimentos weighted tardiness: Resultados en instancias del SDST-JSP. $f =$ 1,3

Inst.	CP				$AG + TS - N_1^S$			
	Mejor	Media	Desv	T(s)	Mejor	Media	Desv	T(s)
10×5								
t2-ps01	4854(13)	4993.7	140.2	60	4454(30)	**4454.0**	0.0	43
t2-ps02	3780(4)	4142.5	157.5	60	3432(30)	**3432.0**	0.0	47
t2-ps03	4380(4)	4608.8	148.4	60	3999(29)	**4000.6**	8.8	51
t2-ps04	3876(4)	4020.8	114.2	60	3732(30)	**3732.0**	0.0	50
t2-ps05	4400(1)	4445.2	41.5	60	3795(26)	**3805.8**	35.9	46
15×5								
t2-ps06	10077(10)	10533.3	415.9	150	9799(5)	**9941.2**	89.2	121
t2-ps07	10439(12)	10552.0	137.5	150	9362(3)	**9507.8**	110.6	122
t2-ps08	10337(2)	10834.2	274.6	150	9660(2)	**9901.8**	134.8	119
t2-ps09	10951(2)	11569.0	305.3	150	9956(2)	**9997.5**	30.0	122
t2-ps10	10985(1)	11999.4	555.8	150	10483(6)	**10569.4**	63.0	119
20×5								
t2-ps11	23176(1)	26168.5	1515.0	300	22406(13)	**23052.3**	692.7	228
t2-ps12	24231(1)	25331.3	675.4	300	22526(2)	**23157.5**	378.3	242
t2-ps13	24714(1)	25728.6	602.4	300	23394(1)	**24025.5**	329.8	247
t2-ps14	26482(1)	27568.7	577.0	300	24928(1)	**25416.0**	399.2	250
t2-ps15	25801(1)	27143.7	775.1	300	24703(3)	**25427.2**	457.0	237
20×15								
ABZ7sdst	23568(1)	26149.8	1343.7	1000	22943(1)	**24466.9**	659.9	754
ABZ8sdst	23011(1)	**24269.9**	598.5	1000	22763(1)	24273.6	721.1	815
ABZ9sdst	22678(1)	24550.4	966.4	1000	21747(1)	**24257.0**	1026.9	787
15×10								
LA21sdst	8961(5)	9470.8	515.9	300	8741(19)	**8829.3**	132.6	220
LA24sdst	9467(1)	9923.4	158.2	300	8501(1)	**8911.7**	147.8	225
LA25sdst	9240(1)	9971.1	484.8	300	8648(18)	**8726.1**	188.8	231
20×10								
LA27sdst	27598(1)	29431.0	884.5	600	26976(1)	**27861.0**	358.1	477
LA29sdst	28444(1)	29602.6	873.1	600	25039(1)	**26928.3**	1109.9	475
15×15								
LA38sdst	9314(1)	9846.0	221.3	450	7317(2)	**8166.7**	437.9	343
LA40sdst	8192(1)	8940.3	336.0	450	7750(1)	**8327.4**	288.8	342

Tabla 10.12: Experimentos weighted tardiness: Resultados en instancias del SDST-JSP. $f =$ 1,5

Inst.	CP				$AG + TS - N_1^S$			
	Mejor	Media	Desv	T(s)	Mejor	Media	Desv	T(s)
10×5								
t2-ps01	3767(12)	3911.0	126.0	60	3342(28)	**3360.7**	71.3	49
t2-ps02	2908(9)	2957.1	127.7	60	2674(30)	**2674.0**	0.0	47
t2-ps03	3420(1)	3559.7	95.4	60	3120(30)	**3120.0**	0.0	50
t2-ps04	2978(1)	3049.9	39.3	60	2890(30)	**2890.0**	0.0	46
t2-ps05	3353(1)	3532.1	115.7	60	2989(8)	**2995.5**	6.5	45
15×5								
t2-ps06	8413(2)	8997.4	396.1	150	8186(22)	**8238.0**	103.4	121
t2-ps07	8734(12)	8875.0	177.9	150	7927(6)	**8079.3**	95.2	123
t2-ps08	8757(1)	9316.7	261.4	150	8182(2)	**8360.1**	88.1	119
t2-ps09	8961(1)	9696.5	447.6	150	8116(1)	**8215.2**	19.0	127
t2-ps10	9177(1)	9967.9	542.1	150	8673(6)	**8744.5**	49.6	115
20×5								
t2-ps11	21083(1)	24131.5	1455.0	300	20285(8)	**20816.4**	471.4	229
t2-ps12	21362(1)	22308.5	494.3	300	20487(1)	**21118.7**	389.8	241
t2-ps13	21912(1)	23547.5	727.2	300	20935(1)	**21821.0**	494.2	243
t2-ps14	24153(1)	25580.0	594.7	300	22543(2)	**23051.2**	224.1	253
t2-ps15	23321(1)	25450.2	1017.3	300	22368(2)	**23028.4**	486.0	238
20×15								
ABZ7sdst	20645(1)	22425.1	803.0	1000	19697(1)	**21350.9**	625.6	754
ABZ8sdst	20047(1)	21049.8	476.2	1000	19437(1)	**20793.0**	759.7	809
ABZ9sdst	19199(1)	**20763.5**	992.0	1000	18778(1)	20924.6	1059.1	782
15×10								
LA21sdst	5893(1)	6888.4	620.9	300	5868(1)	**5959.7**	76.4	214
LA24sdst	6104(3)	6814.8	332.7	300	5941(2)	**6049.1**	88.9	224
LA25sdst	6308(1)	7494.9	607.8	300	5435(23)	**5470.5**	193.5	219
20×10								
LA27sdst	22954(1)	24877.5	1173.1	600	22348(1)	**23383.5**	371.0	479
LA29sdst	22318(1)	24363.7	823.1	600	20833(1)	**22513.4**	890.5	485
15×15								
LA38sdst	4898(1)	5693.3	600.0	450	3344(1)	**3985.2**	365.2	336
LA40sdst	4745(1)	5929.7	733.5	450	4026(2)	**4287.2**	212.7	346

Tabla 10.13: Experimentos weighted tardiness: Resultados en instancias del SDST-JSP. $f =$ 1,6

Inst.	CP				$AG + TS - N_1^S$			
	Mejor	Media	Desv	T(s)	Mejor	Media	Desv	T(s)
10×5								
t2-ps01	3275(4)	3505.8	106.3	60	2852(30)	**2852.0**	0.0	50
t2-ps02	2394(9)	2557.9	166.6	60	2301(30)	**2301.0**	0.0	49
t2-ps03	2894(3)	3142.6	191.8	60	2677(30)	**2677.0**	0.0	51
t2-ps04	2602(5)	2632.4	15.4	60	2538(21)	**2538.9**	1.4	47
t2-ps05	2860(1)	3037.5	59.2	60	2570(12)	**2619.6**	41.8	45
15×5								
t2-ps06	7574(2)	8147.8	431.6	150	7359(15)	**7435.5**	97.3	120
t2-ps07	7982(5)	8641.6	469.2	150	7248(3)	**7424.9**	70.0	121
t2-ps08	8259(1)	8520.7	144.4	150	7524(2)	**7623.9**	70.4	122
t2-ps09	7757(3)	8812.8	489.5	150	7317(29)	**7317.2**	1.1	122
t2-ps10	8420(3)	8848.0	270.2	150	7850(1)	**7914.0**	73.2	113
20×5								
t2-ps11	20170(1)	22908.6	1169.2	300	19239(13)	**19763.8**	566.5	230
t2-ps12	19988(1)	21038.7	469.6	300	19399(1)	**20106.0**	417.0	241
t2-ps13	21315(1)	22278.8	599.2	300	19817(1)	**20618.3**	478.0	243
t2-ps14	22590(1)	24079.1	752.8	300	21190(1)	**21891.7**	368.5	247
t2-ps15	22051(1)	23919.1	992.8	300	21290(2)	**22049.4**	437.7	235
20×15								
ABZ7sdst	19416(1)	20720.9	887.4	1000	18350(1)	**19728.8**	566.8	745
ABZ8sdst	18246(1)	19236.4	622.7	1000	17667(1)	**19159.2**	546.3	807
ABZ9sdst	18607(1)	19632.9	633.8	1000	17256(1)	**19235.3**	1033.3	793
15×10								
LA21sdst	4728(1)	5443.8	455.4	300	4514(3)	**4657.0**	154.4	221
LA24sdst	5513(2)	5956.0	264.4	300	4674(1)	**4831.4**	70.6	222
LA25sdst	4888(1)	5929.9	648.8	300	4105(14)	**4129.3**	96.4	215
20×10								
LA27sdst	21015(1)	22750.5	926.5	600	20047(1)	**21051.2**	397.5	477
LA29sdst	20440(1)	22014.1	802.0	600	19185(1)	**21009.1**	768.2	490
15×15								
LA38sdst	3040(1)	3920.1	408.6	450	1856(4)	**2272.4**	250.7	323
LA40sdst	3025(1)	3855.0	454.7	450	2191(2)	**2550.4**	199.5	344

10.5. Conclusiones

Hemos comenzado este estudio experimental buscando la configuración óptima para nuestros algoritmos, y hemos visto que la mejor opción sigue siendo la combinación de algoritmo genético y búsqueda tabú. Pero en este caso, dado que las estimaciones son mucho menos precisas, comprobamos que es más eficiente a igualdad de tiempos de ejecución calcular el weighted tardiness exacto de todos los vecinos cuya estimación es menor que el weighted tardiness del individuo original. También hemos comprobado que el hecho de utilizar un único camino crítico o todos los caminos críticos proporciona los mejores resultados en diferentes instancias. Por ello hemos decidido elegir de forma aleatoria el número de caminos que utilizaremos en cada una de las búsquedas locales realizadas. Experimentalmente ha resultado ser un método más robusto que cualquiera de las dos alternativas. Hemos experimentado además con las dos estructuras de vecindad propuestas en esta tesis, obteniendo en general N_1^S resultados ligeramente mejores que N^S.

Posteriormente hemos evaluado el algoritmo propuesto comparándolo con los mejores conocidos del estado del arte. Hemos utilizado primero instancias del JSP clásico y nos hemos comparado con el algoritmo de ramificación y poda propuesto por Singer y Pinedo en [132], el algoritmo de búsqueda local de tipo large step random walk (LSRW) propuesto por Kreipl en [79], el algoritmo genético híbrido propuesto por Essafi et al. en [49], y el método de búsqueda local propuesto por Mati et al. en [86]. Los resultados muestran que nuestro algoritmo es muy eficiente y es capaz de superar a los métodos con los que nos hemos comparado, utilizando tiempos de ejecución comparables, a pesar de que nuestro método no esta optimizado para tratar con tiempos de setup nulos. Por otra parte, en el SDST-JSP hemos considerado dos conjuntos de instancias y comparamos nuestros resultados a los obtenidos por el ILOG CPLEX CP Optimizer, debido a que no hemos podido compararnos con el único algoritmo que conocemos de la literatura, por no disponer ni de su código ni de las instancias utilizadas en su estudio experimental. Los resultados del estudio experimental muestran que los resultados de nuestro algoritmo genético híbrido mejoran a los obtenidos por el ILOG CPLEX CP Optimizer.

Capítulo 11

ESTUDIO EXPERIMENTAL. MINIMIZACIÓN DEL TOTAL FLOW TIME

11.1. Introducción

El objetivo de este estudio experimental es comparar nuestros métodos de minimización del total flow time con otros algoritmos del estado del arte. Realizamos en primer lugar experimentos previos para elegir la estructura de vecindad utilizada y el número de caminos críticos considerados. Posteriormente comparamos nuestro algoritmo con métodos que minimizan el total flow time en instancias del JSP clásico. Por último, debido a que no conocemos trabajos referidos al total flow time en el SDST-JSP, nos hemos comparado con los resultados obtenidos por el ILOG CPLEX CP Optimizer.

11.2. Reducción del número de vecinos

Ya hemos comprobado en la sección 10.3.1 que en el weighted tardiness es mucho más eficiente evaluar de forma exacta todos los vecinos cuya estimación es menor que el weighted tardiness del individuo original. Sin embargo las diferencias entre utilizar N_1^S y N^S, o entre

utilizar un camino crítico o todos los caminos críticos eran mucho más sutiles, y este hecho nos motiva a repetir esos experimentos para el caso de la minimización del total flow time.

Para realizar esta comparación hemos utilizado como benchmark las instancias del JSP clásico LA06 a LA20, por dos motivos principalmente. El primero es que es uno de los benchmarks que utilizaremos para compararnos con otros métodos del estado del arte, y el segundo que las instancias tienen tamaños bastante variados, en concreto 15×5 para las LA06-10, 20×5 para las LA11-15, y 10×10 para las LA16-20.

La tabla 11.1 muestra los resultados de estos experimentos. La primera columna indica el nombre de la instancia. La segunda y tercera columnas contienen los resultados utilizando N_1^S y considerando un único camino crítico; en concreto el mejor total flow time y el total flow time medio obtenido en 30 ejecuciones. En la cuarta y quinta columnas se utiliza de nuevo N_1^S pero esta vez considerando todos los caminos críticos de la planificación. Las dos siguientes columnas muestran los resultados utilizando N^S con un único camino crítico, y las dos últimas columnas N^S con todos los caminos críticos.

Los parámetros utilizados en los diferentes experimentos son los indicados en la parte superior de la tabla, en el formato habitual (/tamaño de la población/número de generaciones/máximo número de iteraciones sin mejora en la búsqueda tabú/). Ajustando los parámetros de la forma indicada, el tiempo medio de una única ejecución ha sido de 324, 312, 415 y 366 segundos para cada una de las cuatro configuraciones elegidas, en el mismo orden en el que están en la tabla 11.1. Hemos elegido los tiempos de ejecución para que sean comparables a los que utilizan los métodos del estado del arte con los que nos compararemos más adelante. Por otra parte, hemos marcado en negrita las medias de total flow time que son iguales o mejores al del resto de configuraciones.

Recordemos que en la minimización del weighted tardiness, en algunas instancias ofrecía mejores resultados considerar un único camino crítico y en otras instancias todos los caminos críticos. Por el contrario, en esta serie de experimentos ha resultado ser mucho más eficiente utilizar todos los caminos críticos para todas las instancias consideradas. N_1^S con todos los caminos críticos mejora a N_1^S con un único camino crítico, mientras que N^S con todos los caminos críticos es superior a N^S con un único camino crítico.

Por otra parte, las diferencias entre utilizar N_1^S con todos los caminos críticos y N^S con todos los caminos críticos en general no son muy grandes. De todas formas, la media de total flow time es, en todos los casos, menor o igual con la estructura N_1^S, y además el tiempo de ejecución medio es más reducido (312 segundos de media contra 366 segundos de

Tabla 11.1: Experimentos total flow time: Reducción del número de vecinos

	N_1^S Uno /100/180/250/		N_1^S Todos /100/150/100/		N^S Uno 120/160/50/		N^S Todos /70/100/50/	
Inst.	Mejor	Media	Mejor	Media	Mejor	Media	Mejor	Media
LA06	8632(14)	8652	8625(8)	**8629**	8632(22)	8645	8625(6)	8630
LA07	8116(1)	8196	8069(30)	**8069**	8109(1)	8170	8069(18)	8077
LA08	7946(1)	8017	7946(23)	**7948**	7949(1)	7991	7946(24)	7953
LA09	9077(4)	9153	9034(1)	**9063**	9034(1)	9114	9034(1)	9066
LA10	8799(7)	8867	8798(16)	**8799**	8799(6)	8831	8798(10)	8801
LA11	13995(1)	14133	13836(1)	**13958**	13902(1)	14074	13927(1)	13993
LA12	11710(1)	11802	11710(4)	**11728**	11702(1)	11786	11710(2)	11736
LA13	13408(1)	13512	13280(1)	**13339**	13408(1)	13503	13281(1)	13344
LA14	14616(1)	14697	14464(1)	**14543**	14546(1)	14664	14452(1)	14558
LA15	14130(2)	14283	14111(1)	**14156**	14130(1)	14257	14130(1)	14175
LA16	7376(26)	7378	7376(30)	**7376**	7376(29)	7377	7376(30)	**7376**
LA17	6537(1)	6567	6537(30)	**6537**	6537(1)	6560	6537(30)	**6537**
LA18	7010(7)	7097	6970(30)	**6970**	7010(5)	7083	6970(24)	6982
LA19	7218(1)	7227	7217(27)	**7218**	7217(2)	7243	7217(24)	7219
LA20	7450(2)	7515	7345(16)	**7382**	7450(2)	7495	7345(7)	7417

media). Por estos dos motivos esa es la configuración elegida para el resto de este estudio experimental.

11.3. Minimización del total flow time en el JSP

Para comparar nuestro algoritmo con otros del estado del arte, consideraremos el método exacto A^* combinado con un método de poda por dominancia, denominado A^*-PD y una variante subóptima de este algoritmo que utiliza un método de ponderación heurística, denominado A^*-DW. Ambos métodos son explicados con detalle por Sierra en [125] y también, de forma más resumida, en esta misma tesis en la sección 4.4.2. Además, también consideraremos el algoritmo de búsqueda local de tipo *Large Step Random Walk* (LSRW) propuesto por Kreipl en [79]. Ese algoritmo en principio está diseñado para la minimización del weighted tardiness, pero recordemos que esa función objetivo se puede reducir al total flow time sin más que definir todos los due dates iguales a 0, y todos los pesos de los trabajos iguales a 1. Para poder compararnos con LSRW, hemos utilizado la implementación incluída en la herramienta LEKIN®, que se puede descargar en

http://www.stern.nyu.edu/om/software/lekin/index.htm.

En este estudio experimental proponemos una combinación del algoritmo genético descrito en el capítulo 5 con la búsqueda tabú descrita en la sección 6.4. No consideraremos la gestión de la lista de soluciones elite para la búsqueda tabú, ya que al combinarla con el genético utilizaremos un número máximo de iteraciones bastante reducido para que el tiempo de ejecución no sea excesivo. A cada cromosoma generado por el algoritmo genético se le aplicará el constructor de planificacones SSGS, descrito en la sección 3.3.2. Por otra parte utilizamos la estructura de vecindad N_1^S y consideramos todos los caminos críticos de cada planificación, ya que es la configuración que mejores resultados obtuvo en la sección anterior. Denominaremos a partir de este momento $AG + TS - N_1^S$ al algoritmo propuesto para este estudio experimental.

Hemos realizado dos series de experimentos en instancias de benchmarks estándar elegidos de la OR-library. En la primera serie experimentamos sobre el conjunto de instancias (de tamaños 10×5, 8×8 y 9×9) propuestas por Sierra en [125], y que son resueltas de forma óptima por el algoritmo A^*-PD. Las instancias reducidas de tamaño 8×8 y 9×9 se construyen a partir de las instancias 10×10 originales quitando los últimos trabajos y las últimas máquinas. En la segunda serie de experimentos utilizamos instancias más grandes (de tamaños 15×5, 20×5 y 10×10) que no son resueltas de forma óptima por A^*-PD. En todos estos experimentos debemos tener en cuenta la diferencia entre las máquinas y el tiempo de ejecución. Las versiones del A^* se ejecutan en Ubuntu V8.04 en un Intel Core 2 Duo a 2.13GHz con 7.6Gb de RAM, LSRW se ejecuta en Windows XP en un Intel Core 2 Duo a 2.13GHz con 3Gb de RAM, y $AG + TS - N_1^S$ se ejecuta en Windows XP en un Intel Core 2 Duo a 2.66GHz con 2Gb de RAM.

La tabla 11.2 muestra los resultados de la primera serie de experimentos. En este caso A^*-PD alcanza la solución óptima en todas las instancias. LSRW y $AG + TS - N_1^S$ se han ejecutado 20 veces para cada instancia y se indica la mejor solución obtenida, la solución media y el tiempo medio de una única ejecución. Los parámetros de $AG + TS - N_1^S$ (/ tamaño de la población de AG / número de generaciones de AG / máximas iteraciones sin mejora de TS /) son /70/90/100/ para las instancias de tamaño 10×5 y 9×9, y /30/40/50/ para las instancias de tamaño 8×8. Con estos valores el algoritmo converge de manera apropiada y el tiempo de ejecución no es mayor que el de los otros dos algoritmos, teniendo en cuenta en la medida de lo posible la diferencia entre las máquinas.

Tabla 11.2: Experimentos total flow time: Comparación con el estado del arte en el JSP

Inst.	A^*-PD		LSRW			$AG + TS - N_1^S$		
	Óptimo	T(s)	Mejor	Media	T(s)	Mejor	Media	T(s)
Instancias 10×5								
LA01	4832	35	4832(1)	4832.9	93	4832(20)	4832.0	43
LA02	4459	80	4479(4)	4483.2	93	4459(17)	4460.4	44
LA03	4151	10	4151(20)	4151.0	93	4151(20)	4151.0	41
LA04	4259	19	4259(2)	4268.8	93	4259(20)	4259.0	41
LA05	4072	68	4072(2)	4095.0	93	4072(20)	4072.0	45
Instancias 8×8								
LA16	4600	4	4600(3)	4606.5	17	4600(20)	4600.0	4
LA17	4366	6	4379(20)	4379.0	17	4366(20)	4366.0	4
LA18	4690	3	4690(13)	4704.7	17	4690(4)	4723.6	4
LA19	4612	3	4612(20)	4612.0	17	4612(19)	4613.8	4
LA20	4616	5	4616(20)	4616.0	17	4616(20)	4616.0	3
ORB01	4743	4	4743(20)	4743.0	17	4743(20)	4743.0	4
ORB02	4678	5	4678(20)	4678.0	17	4678(20)	4678.0	4
ORB03	4925	10	4925(20)	4925.0	17	4925(20)	4925.0	4
ORB04	5081	4	5081(20)	5081.0	17	5081(17)	5096.2	4
ORB05	4191	3	4191(5)	4192.5	17	4191(20)	4191.0	4
ORB06	4673	11	4673(20)	4673.0	17	4673(20)	4673.0	4
ORB07	2124	9	2124(20)	2124.0	17	2124(19)	2125.5	4
ORB08	4749	40	4749(6)	4759.9	17	4749(10)	4757.9	4
ORB09	4590	20	4590(20)	4590.0	17	4590(19)	4590.5	4
ORB10	4959	1	4959(20)	4959.0	17	4959(20)	4959.0	4
ABZ5	6818	3	6839(4)	6891.0	17	6818(13)	6828.7	4
ABZ6	4900	4	4900(2)	4922.5	17	4900(20)	4900.0	4
FT10	4559	4	4559(20)	4559.0	17	4559(19)	4562.1	4
Instancias 9×9								
LA16	5724	38	5724(6)	5739.6	294	5724(20)	5724.0	49
LA17	5390	116	5396(5)	5403.5	294	5390(19)	5390.6	53
LA18	5770	34	5770(20)	5770.0	294	5770(20)	5770.0	55
LA19	5891	28	5891(20)	5891.0	294	5891(20)	5891.0	50
LA20	5915	110	5934(12)	5935.2	294	5915(17)	5917.9	51
ORB01	6367	166	6367(5)	6378.5	294	6367(13)	6369.1	61
ORB02	5867	92	5867(3)	5867.9	294	5867(7)	5869.3	51
ORB03	6310	110	6310(20)	6310.0	294	6310(20)	6310.0	54
ORB04	6661	273	6661(20)	6661.0	294	6661(5)	6674.5	58
ORB05	5605	16	5605(20)	5605.0	294	5605(20)	5605.0	52
ORB06	6106	208	6106(20)	6106.0	294	6106(20)	6106.0	49
ORB07	2668	155	2668(20)	2668.0	294	2668(20)	2668.0	56
ORB08	5656	772	5668(2)	5693.3	294	5668(18)	5672.9	60
ORB09	6013	38	6013(18)	6013.8	294	6013(14)	6016.9	58
ORB10	6328	106	6328(1)	6332.8	294	6328(20)	6328.0	57
ABZ5	8586	39	8586(20)	8586.0	294	8586(20)	8586.0	51
ABZ6	6524	29	6524(14)	6524.6	294	6524(20)	6524.0	49
FT10	5982	72	5982(20)	5982.0	294	5982(20)	5982.0	52

Como se puede observar, $AG + TS - N_1^S$ ha sido capaz de alcanzar la solución óptima en la mayoría de las ejecuciones: en concreto en 712 de las 820 ejecuciones para las 41 instancias (86.8 %), y ha alcanzado por lo menos una vez la solución óptima en 40 de las 41 instancias. La excepción es la instancia ORB08(9×9) en donde A^*-PD necesita el mayor tiempo para encontrar el óptimo de entre todas las instancias 9×9, un tiempo 12 veces superior al utilizado por $AG + TS - N_1^S$. Por otra parte LSRW ha sido capaz de alcanzar la solución óptima en 501 de las 820 ejecuciones (61.1 %), y no ha sido capaz de encontrar la solución óptima en 6 de las 41 instancias. Por otra parte, si calculamos los errores relativos de las medias de total flow time respecto a la solución óptima de la instancia, vemos que el error relativo medio para $AG + TS - N_1^S$ es de 0.056 %, mientras que el error relativo medio para LSRW es de 0.140 %. También podemos observar que el tiempo de ejecución empleado por $AG + TS - N_1^S$ no depende mucho de la instancia en particular, al contrario de lo que ocurre con A^*-PD.

En la segunda serie de experimentos consideramos los conjuntos de instancias LA06-10 (15×5), LA11-15 (20×5) y LA15-20 (10×10). Ninguna de estas instancias se puede resolver de forma óptima con A^*-PD antes de agotar la memoria de la máquina, y por lo tanto no conocemos sus soluciones óptimas. Por este motivo, en estos casos utilizamos para compararnos la variante subóptima A^*-DW. También nos compararemos, de nuevo, con el método LSRW propuesto en [79]. Los parámetros de $AG + TS - N_1^S$ en estos experimentos son /100/150/100/ para las 15 instancias.

La tabla 11.3 muestra los resultados. Aquí, el tiempo indicado para A^*-DW corresponde a la suma de varias ejecuciones que son necesarias para ajustar el parámetro δ a su mejor valor, y una ejecución más para obtener todas las posibles soluciones con ese parámetro ya ajustado (ver sección 4.4.2). Para LSRW y $AG + TS - N_1^S$ el tiempo indicado es el tiempo medio que dura una de las ejecuciones. Debemos notar que en esta serie de experimentos hemos realizado 20 ejecuciones para cada instancia con LSRW, mientras que con $AG + TS - N_1^S$ hemos realizado 30 ejecuciones para cada instancia, por lo que en este caso LSRW se encuentra en ligera desventaja en cuanto al mejor total flow time alcanzado entre todas las ejecuciones.

En la tabla marcamos en negrita las medias de total flow time que igualan o superan a todos los demás métodos. Si comparamos $AG + TS - N_1^S$ con LSRW podemos observar que el resultado medio es mejor en 14 instancias y peor en 1; y el mejor valor de nuestro algoritmo es mejor que el obtenido por LSRW en 11 instancias e igual en 4. Además, los

Tabla 11.3: Experimentos total flow time: Comparación con el estado del arte en el JSP. Segundo conjunto de instancias

Inst.	Tamaño	A^*-DW		LSRW			$AG+TS-N_1^S$		
		Mejor	T(s)	Mejor	Media	T(s)	Mejor	Media	T(s)
LA06	15×5	8631	859	8644(1)	8670.9	840	8625(8)	**8629.4**	242
LA07	15×5	**8069**	1005	8116(1)	8165.9	840	8069(30)	**8069.0**	233
LA08	15×5	8190	732	7949(8)	7960.7	840	7946(23)	**7948.4**	258
LA09	15×5	9153	583	9113(1)	9186.6	840	9034(1)	**9063.0**	267
LA10	15×5	**8798**	763	8821(1)	8881.7	840	8798(16)	8799.0	234
LA11	20×5	14014	657	14148(2)	14196.4	840	13836(1)	**13957.8**	544
LA12	20×5	12594	501	11733(1)	11819.0	840	11710(4)	**11728.1**	507
LA13	20×5	13495	538	13477(1)	13558.1	840	13280(1)	**13338.6**	555
LA14	20×5	14556	595	14671(1)	14738.7	840	14464(1)	**14542.5**	476
LA15	20×5	14279	519	14285(1)	14380.0	840	14111(1)	**14155.6**	539
LA16	10×10	**7376**	2143	7376(19)	7376.5	840	7376(30)	**7376.0**	158
LA17	10×10	**6537**	2439	6537(1)	6566.8	840	6537(30)	**6537.0**	154
LA18	10×10	**6970**	3829	6970(1)	7005.0	840	6970(30)	**6970.0**	175
LA19	10×10	**7217**	1503	7217(15)	7217.7	840	7217(27)	7218.0	178
LA20	10×10	**7345**	1351	7394(15)	7397.4	840	7345(16)	7381.5	161

tiempos de ejecución utilizados por $AG+TS-N_1^S$ son menores en todos los casos. Por otra parte, si lo comparamos con A^*-DW, nuestro algoritmo es en media mejor en 8 casos, igual en 4 y peor en 3, y el mejor valor supera al obtenido por A^*-DW en 8 instancias y es igual en todas las restantes. Además los tiempos de ejecución del algoritmo genético híbrido son en general más reducidos, especialmente en las instancias de tamaño 10×10.

Por todas estas razones podemos concluir que $AG + TS - N_1^S$ es un método más eficiente que los otros mostrados, o como mínimo competitivo con ellos, utilizando tiempos de ejecución por lo general más reducidos.

11.4. Minimización del total flow time en el SDST-JSP

Para minimizar el total flow time en el SDST-JSP, hemos considerado las instancias del conjunto BT y el conjunto de instancias propuesto por Vela et al. en [151]. Los detalles de estos benchmarks se pueden consultar en las secciones 8.2.1 y 8.2.3, respectivamente. Debido a que no conocemos trabajos que minimicen esta función objetivo en el SDST-JSP, hemos decidido modelar el problema con el ILOG CPLEX CP Optimizer (CP) y comparar

nuestros resultados con los obtenidos por dicho método. Los detalles del método CP se pueden consultar en la sección 4.4.3.

La configuración utilizada para estos experimentos es de 100 cromosomas en la población, 150 generaciones y un máximo de 100 iteraciones sin mejora para la búsqueda tabú, es decir la misma configuración utilizada en la segunda serie de experimentos de la sección anterior. Hemos realizado 30 ejecuciones para cada una de las instancias tanto con $AG + TS - N_1^S$ como con CP. Por otra parte, hemos dejado a CP un tiempo de ejecución aproximadamente 20 % superior al utilizado por $AG + TS - N_1^S$.

En la tabla 11.4 presentamos los resultados de esta serie de experimentos. Indicamos, para cada uno de los dos métodos, el mejor total flow time alcanzado y el número de veces que se alcanza, el total flow time medio, la desviación estándar y el tiempo medio consumido por una única ejecución.

Observamos que el total flow time medio obtenido por $AG + TS - N_1^S$ es mejor en 24 de las 25 instancias al obtenido por CP, con la única excepción de la instancia ABZ7sdst. Hay que destacar que los resultados de CP son excelentes, teniendo en cuenta que es un método de resolución general. En término medio CP consigue un total flow time medio un 3.5 % superior al de $AG + TS - N_1^S$. Sin embargo, las diferencias entre los dos métodos son bastante menores que en el caso de los experimentos sobre minimización del weighted tardiness mostrados en la sección 10.4, ya que recordemos que en aquel caso CP obtuvo en media un valor de la función objetivo el 13.6 % superior. Esto puede ser debido a que para el total flow time hemos dado un mayor tiempo de ejecución a los dos métodos y eso ha igualado más los resultados, aunque también puede ser debido a que CP se comporta mejor con esta función objetivo que con la otra. Las menores diferencias se aprecian en las instancias de tamaño 20×15, en las cuales el total flow time medio obtenido por CP es únicamente un 0.3 % mayor en media, mientras que las mayores diferencias se aprecian en las de tamaño 15×10, en las que CP obtiene un total flow time medio un 5.5 % mayor. En cuanto a los mejores total flow time alcanzados, $AG + TS - N_1^S$ alcanza uno mejor que CP en 19 instancias, igual en 4 instancias (todas ellas de tamaño 10×5), y peor en 2 instancias (ABZ7sdst y ABZ9sdst). Podemos concluir, entonces, que $AG + TS - N_1^S$ supera a CP en esta serie de experimentos.

Tabla 11.4: Experimentos total flow time: Resultados en instancias del SDST-JSP

Inst.	CP				$AG + TS - N_1^S$			
	Mejor	Media	Desv	T(s)	Mejor	Media	Desv	T(s)
10×5								
t2-ps01	6050(3)	6173.3	136.1	130	6050(11)	**6053.8**	2.9	101
t2-ps02	5505(7)	5649.5	107.2	130	5505(30)	**5505.0**	0.0	107
t2-ps03	5092(9)	5176.2	139.1	130	5092(30)	**5092.0**	0.0	102
t2-ps04	5512(2)	5603.6	65.0	130	5478(19)	**5499.5**	29.6	111
t2-ps05	5041(1)	5173.3	63.0	130	5041(27)	**5045.6**	14.0	106
15×5								
t2-ps06	10795(1)	11135.0	223.2	330	10580(2)	**10684.7**	80.8	261
t2-ps07	10348(6)	10418.1	68.7	330	10007(4)	**10064.5**	35.7	277
t2-ps08	10418(1)	10705.5	198.2	330	10096(4)	**10147.4**	30.9	270
t2-ps09	10993(9)	11285.1	243.2	330	10835(10)	**10908.6**	69.4	271
t2-ps10	10785(1)	11112.7	216.8	330	10692(8)	**10737.6**	51.8	259
20×5								
t2-ps11	19475(1)	20504.6	345.5	660	19110(3)	**19440.2**	280.3	519
t2-ps12	18793(1)	19151.5	288.6	660	18027(1)	**18327.2**	196.5	519
t2-ps13	19856(1)	20385.4	179.3	660	19183(1)	**19864.0**	312.8	548
t2-ps14	20955(1)	21153.2	121.4	660	20135(1)	**20448.7**	248.8	528
t2-ps15	21088(1)	21775.5	421.4	660	20298(1)	**20592.2**	170.8	532
20×15								
ABZ7sdst	21232(1)	**22087.2**	312.6	2000	21508(1)	22094.0	305.3	1560
ABZ8sdst	21823(1)	22332.9	295.9	2000	21334(1)	**22155.1**	317.6	1580
ABZ9sdst	21365(1)	21925.3	338.8	2000	21488(1)	**21869.5**	240.4	1628
15×10								
LA21sdst	15364(1)	16095.1	309.2	600	15089(1)	**15247.7**	69.0	468
LA24sdst	14985(1)	15275.5	130.8	600	14424(6)	**14546.4**	133.4	473
LA25sdst	14703(1)	15390.5	298.2	600	14405(1)	**14525.0**	89.5	475
20×10								
LA27sdst	28664(1)	29595.9	402.2	1200	28115(1)	**28688.9**	257.5	1013
LA29sdst	27480(1)	28437.9	427.3	1200	26335(1)	**26862.3**	297.2	1010
15×15								
LA38sdst	18951(1)	19588.9	327.7	850	18481(1)	**18688.1**	133.1	658
LA40sdst	19545(1)	20056.5	216.7	850	19031(3)	**19178.4**	108.7	697

11.5. Conclusiones

Hemos comenzado este estudio experimental sobre el total flow time comprobando que, para esta función objetivo y con el conjunto de instancias elegido, la mejor configuración es la estructura de vecindad N_1^S considerando todos los caminos críticos de la planificación para generar vecinos. Posteriormente comparamos nuestros métodos para minimizar el total flow time con otros algoritmos del estado del arte. Comenzamos utilizando instancias del JSP tradicional sin tiempos de setup, y en ese caso nos hemos comparado con dos métodos propuestos por Sierra en [125]: el método exacto A^* combinado con un método de poda por dominancia, denominado A^*-PD y una variante subóptima de este algoritmo que utiliza un método de ponderación heurística, denominado A^*-DW. También nos hemos comparado con el algoritmo de búsqueda local de tipo large step random walk (LSRW) propuesto por Kreipl en [79]. Los resultados experimentales muestran que nuestro algoritmo es competitivo con los otros métodos del estado del arte, siendo capaz de conseguir mejores cotas superiores en varias de las instancias de mayor tamaño. Por otra parte, por lo que sabemos no existe literatura sobre minimización del total flow time en el SDST-JSP, por lo que nos hemos comparado en dos conjuntos de instancias con los resultados obtenidos por el método general ILOG CPLEX CP Optimizer, obteniendo mejores resultados que él en la gran mayoría de las instancias.

Capítulo 12

CONCLUSIONES

En esta tesis hemos abordado el Job-Shop Scheduling Problem with Sequence-Dependent Setup Times (SDST-JSP), y hemos tratado la minimización de diversas funciones objetivo para dicho problema. El interés de tratar varias funciones objetivo es que diferentes aplicaciones prácticas pueden requerir diferentes criterios de optimización. En concreto estudiamos las denominadas makespan, maximum lateness, weighted tardiness y total flow time. La función objetivo más estudiada en la literatura es, con diferencia, el makespan. Sin embargo las demás funciones tienen muchas veces más interés en aplicaciones reales.

Los métodos utilizados han sido los algoritmos genéticos y varias formas de búsqueda local. A continuación mostramos un resumen de las principales aportaciones al estado del arte realizadas en esta tesis, las conclusiones obtenidas, las posibles líneas futuras de investigación, y las publicaciones resultantes del trabajo de investigación realizado.

12.1. Conclusiones y Aportaciones

Para el desarrollo de esta tesis hemos elegido los algoritmos genéticos y la búsqueda local porque es bien sabido que son métodos capaces de conseguir soluciones de alta calidad de forma eficiente, incluso en problemas de gran tamaño y complejidad. No obstante, la revisión bibliográfica realizada contempla tanto ese tipo de algoritmos como otros de búsqueda heurística en espacios de estados, shifting bottleneck, reglas de prioridad o programación con restricciones.

Para tratar la minimización de las diferentes funciones objetivo consideradas nos basamos en el modelo de grafo disyuntivo. Proponemos grafos disyuntivos diferentes para tratar las distintas funciones. Esta metodología permite utilizar los mismos métodos para todas ellas sin más que adaptarlos al grafo correspondiente. En función las diferencias estructurales, básicamente según el número de nodos finales, esta adaptación será casi inmediata o precisará de la reformulación de condiciones y algoritmos. Por ejemplo, el grafo propuesto para el maximum lateness permite utilizar los mismos métodos utilizados para el makespan, y ello evita tener que desarrollar métodos completamente nuevos. El grafo disyuntivo para las funciones objetivo de tipo suma, como el weighted tardiness y el total flow time, tiene varios nodos finales y ello implica que su resolución es más compleja, como hemos visto a lo largo de esta tesis.

El algoritmo genético utilizado es una adaptación de un algoritmo genético desarrollado previamente para el JSP, en el que la principal diferencia es el algoritmo de decodificación. Hemos considerado primero una extensión al SDST-JSP del conocido algoritmo $G\&T$ diseñado por Giffler y Thompson para el JSP. Al contrario de lo que ocurre con el algoritmo $G\&T$ para el JSP, esta extensión no genera un conjunto de planificaciones dominante para el SDST-JSP, pero sin embargo es simple y eficiente. También hemos considerado otro constructor de planificaciones activas denominado $SSGS$ (Serial Schedule Generation Schema) que además de generar un conjunto de planificaciones dominante, ha resultado ser más eficiente. Sin embargo hemos visto en los estudios experimentales que la utilización de un constructor de planificaciones semiactivas también es una alternativa muy eficaz.

Además, hemos demostrado que muchas de las propiedades formales ya estudiadas para el JSP clásico, que facilitaban el diseño de algoritmos de resolución, dejan de cumplirse en presencia de tiempos de setup. Así, por ejemplo, una planificación cuyo camino crítico consiste únicamente en tareas de un mismo trabajo o de una misma máquina no necesariamente corresponde a una planificación óptima. Por otra parte, ninguna estructura de vecindad basada en invertir órdenes de procesamiento de tareas críticas puede cumplir la propiedad de conectividad en el SDST-JSP. Además, invertir un único arco crítico puede dar lugar a una planificación no factible, y también puede dar lugar a una planificación mejor incluso aunque dicho arco crítico no se encuentre en ninguno de los extremos de su bloque crítico. Todas estas consideraciones hacen que sea más compleja la resolución del SDST-JSP que la del JSP clásico.

Para desarrollar un método eficiente de búsqueda local, hemos extendido y formalizado la

estructura de vecindad denominada N_1 por Mattfeld en [88], que se basa en la inversión de un único arco crítico en cada movimiento. Hemos estudiado los movimientos que pueden llevar a soluciones que mejoran en presencia de tiempos de setup, y hemos establecido condiciones que permiten descartar planificaciones no factibles o que no mejoran. Hemos denominado N_1^S a esta nueva estructura para el SDST-JSP. También hemos desarrollado otra estructura de tipo inserción, que puede modificar el orden de procesamiento de más de dos tareas en el mismo movimiento, y que denominamos N^S. Para desarrollarla hemos extendido las condiciones de factibilidad y no mejora estudiadas para N_1^S. Por otra parte también hemos propuesto métodos para estimar los vecinos para las diferentes funciones objetivo y hemos estudiado la eficacia de dichos métodos de estimación, llegando a la conclusion de que las funciones objetivo de tipo suma, como el weighted tardiness y el total flow time, son más difíciles de estimar que las funciones objetivo de tipo bottleneck, como el makespan o el maximum lateness. También hemos comprobado en qué casos las estimaciones que ofrecen los métodos son una cota inferior del valor real de la función objetivo.

Hemos explotado las estructuras de vecindad propuestas dentro de diversas estrategias de búsqueda local. Experimentamos con métodos de escalada, con la idea de combinarlos con el algoritmo genético diseñado, ya que dichas combinaciones suelen funcionar mejor que cada uno de los métodos por separado. También proponemos un método de búsqueda tabú que es capaz de obtener muy buenos resultados en un tiempo computacional muy reducido. Por último, hemos experimentado con el algoritmo híbrido que resulta de combinar el algoritmo genético con la búsqueda tabú, y ese ha resultado ser el mejor método de resolución cuando el tiempo de ejecución disponible no es demasiado reducido.

Hemos realizado un estudio experimental para cada una de las funciones objetivo estudiadas, por una parte con el objetivo de encontrar la mejor configuración posible para nuestros métodos, y por otra parte para comparar nuestros mejores resultados con los resultados de los métodos más representativos del estado del arte. En los estudios experimentales realizamos por lo general 30 ejecuciones de nuestros algoritmos sobre cada instancia, ya que los métodos utilizados son estocásticos. En las tablas de resultados proporcionamos el mejor valor obtenido entre todas las ejecuciones, la media del valor obtenido en las ejecuciones, la desviación estándar y el tiempo de computación medio de una ejecución. Con estos datos y ese número de ejecuciones podemos obtener conclusiones bastante fiables. A continuación resumimos brevemente el estudio experimental realizado para cada una de las funciones objetivo.

En cuanto a la minimización del makespan en el SDST-JSP, hemos dividido el estudio experimental en dos partes: en la primera probamos diferentes configuraciones para nuestros métodos y en la segunda nos comparamos con los mejores métodos conocidos. En la primera parte comprobamos que la estructura de vecindad N^S es la que ofrece mejores resultados, y que la combinación de algoritmo genético con búsqueda tabú es mejor que la utilización de cada método por separado, y también mejor que la combinación de un algoritmo genético con una búsqueda local más simple como puede ser la escalada simple o la escalada de máximo gradiente. También mostramos estudios sobre otros aspectos, como pueden ser el constructor de planificaciones utilizado o la presencia de un operador de mutación en el algoritmo genético, por ejemplo.

A continuación comparamos nuestros resultados con los obtenidos por los mejores métodos conocidos en la literatura, utilizando para ello los benchmarks sobre los que los diferentes autores han publicado los resultados de sus métodos. Así, en primer lugar utilizamos el conjunto BT propuesto por Brucker y Thiele en [33]. Las 15 instancias del conjunto BT nos permiten comparar nuestro método con algunos de los mejores del estado del arte, como el algoritmo de ramificación y poda propuesto por Artigues y Feillet en [15] y el método basado en el heurístico shifting bottleneck combinado con búsqueda local guiada propuesto por Balas et al. en [19]. El primero es un método que proporciona la mejor cota inferior conocida para las 15 instancias del conjunto BT, mientras que el segundo ha conseguido las mejores cotas superiores para cuatro de las cinco instancias de mayor tamaño de dicho conjunto. Nuestro algoritmo genético híbrido ha alcanzado la mejor solución conocida en 14 de las 15 instancias, logrando mejorarla en 6 de las 15 instancias: las cinco de mayor tamaño y una de tamaño medio (en las instancias pequeñas no existe mejora posible pues la mejor solución conocida es la solución óptima). Incluso observando el valor de la media, el algoritmo genético híbrido obtiene mejores resultados que el método de ramificación y poda y que el algoritmo shifting bottleneck en cuatro de las cinco instancias grandes. Por lo tanto, podemos concluir que el algoritmo genético híbrido que hemos propuesto es un método heurístico muy eficiente para la minimización del makespan en el SDST-JSP.

También mostramos experimentos sobre el benchmark propuesto por Cheung y Zhou en [40], logrando superar ampliamente los resultados obtenidos en dicho trabajo. Consideramos además el conjunto propuesto por Vela et al. en [151], con instancias más grandes y más difíciles que las del conjunto BT, y nuestro algoritmo genético híbrido es capaz de mejorar los resultados obtenidos por el ILOG CPLEX CP Optimizer. Para concluir la sección sobre la

minimización del makespan también ofrecemos resultados sobre benchmarks del JSP clásico sin tiempos de setup, con resultados también muy competitivos con los mejores métodos conocidos en la literatura.

Para minimizar el maximum lateness en el SDST-JSP, el grafo disyuntivo que modeliza este problema hace que sean trivialmente extensibles a él los métodos de resolución utilizados para la minimización del makespan. En consecuencia, consideramos para la experimentación la estructura de vecindad N^S, ya que es la que mejores resultados obtuvo en el análisis realizado para el makespan. En la minimización del maximum lateness indicamos resultados sobre los benchmarks propuestos por Ovacik y Uzsoy en [109] y comparamos nuestros algoritmos con dos de los métodos más representativos del estado del arte actual: el heurístico shifting bottleneck combinado con búsqueda local guiada (SB-GLS) propuesto por Balas et al. en [20], y el algoritmo de tipo Iterative Sampling Search (ISS) propuesto por Oddi et al. en [107]. En este estudio experimental debemos utilizar un tiempo de ejecución reducido para poder compararnos con los resultados de [20] en igualdad de condiciones, y por este motivo utilizamos una búsqueda tabú por separado, ya que comprobamos que la hibridación con un algoritmo genético era beneficiosa sólo cuando el tiempo de ejecución era lo suficientemente elevado. Los resultados del estudio experimental muestran que nuestro algoritmo mejora a los otros dos métodos, siendo además menos sensible a los cambios en los parámetros que definen las instancias. Además, nuestro algoritmo establece nuevas mejores soluciones para la mayoría de las instancias de este benchmark, y ello puede servir como referencia a futuras investigaciones.

Hemos considerado también la minimización del weighted tardiness en el SDST-JSP, definiendo una representación de grafo disyuntivo para este problema. En el caso de las funciones objetivo de tipo suma no hemos propuesto ni demostrado condiciones de no mejora para las estructuras de vecindad utilizadas, ya que la utilización de dichas condiciones en funciones de este tipo es mucho más compleja. Sí hemos definido, en cambio, condiciones de factibilidad adaptadas y un método para estimar el weighted tardiness de los vecinos, si bien hemos constatado que estimar esta función objetivo es más difícil que estimar otras funciones objetivo clásicas como el makespan. En el capítulo del estudio experimental sobre la minimización del weighted tardiness utilizamos las dos estructuras de vecindad propuestas en esta tesis en un método de búsqueda tabú que se aplica a los cromosomas generados por un algoritmo genético. Hemos comprobado que la opción que ofrece mejores resultados es calcular de forma exacta la función objetivo para todos los vecinos cuya estimación sea

menor que la función objetivo del individuo original. Por el contrario, en la minimización del makespan era más conveniente elegir simplemente el vecino con la menor estimación, debido a que en las funciones objetivo de tipo bottleneck el algoritmo de estimación es mucho más preciso. También hemos comprobado que, dependiendo de la instancia en concreto, puede ser mejor considerar un único camino crítico o todos los caminos críticos al generar vecinos. Por ello hemos decidido elegir de forma aleatoria el número de caminos que utilizaremos en cada una de las búsquedas locales, con la idea de obtener simultáneamente los beneficios de las dos estrategias. En los experimentos ha resultado ser un método más robusto que cualquiera de las dos alternativas por separado. En cuanto a la estructura de vecindad, tanto N_1^S como N^S obtienen resultados sobresalientes, sin embargo en el caso de esta función objetivo la estructura más simple N_1^S obtiene en general resultados ligeramente mejores que N^S.

De nuevo el objetivo es comparar nuestros métodos con otros algoritmos del estado del arte. Primero hemos utilizado benchmarks del JSP clásico sin tiempos de setup, y comparamos nuestros resultados con los obtenidos por el algoritmo de búsqueda local de tipo large step random walk (LSRW) propuesto por Kreipl en [79], el algoritmo genético híbrido propuesto por Essafi et al. en [49], y el método de búsqueda local propuesto por Mati et al. en [86]. Utilizamos en el estudio experimental dos conjuntos de instancias: el primero de ellos es el conocido benchmark propuesto por Singer y Pinedo en [132], y el segundo es el benchmark propuesto por Essafi et al. en [49], con instancias de un tamaño en general mayor. Los resultados del estudio experimental muestran que nuestro algoritmo es competitivo con los otros métodos del estado del arte. En cuanto al SDST-JSP, el único algoritmo que conocemos que minimice el weighted tardiness en este problema es el heurístico shifting bottleneck propuesto por Sun y Noble en [139]. El problema es que en el estudio experimental realizado en ese trabajo se utilizan instancias generadas de forma aleatoria, y los autores ya no disponen ni del código de su algoritmo ni de las instancias utilizadas. Debido a este problema, en el SDST-JSP optamos por modelar el problema mediante el ILOG CPLEX CP Optimizer y comparar con él nuestro algoritmo. En la serie de experimentos realizada se observa que nuestro algoritmo supera los resultados obtenidos por el otro método.

Por último, tratamos la minimización del total flow time en el SDST-JSP. Como esta función objetivo es realmente un caso particular del weighted tardiness, la representación de grafo disyuntivo utilizada, las estructuras de vecindad y el algoritmo de estimación de vecinos utilizado son similares a las utilizadas en la otra función objetivo. Para el estudio experimental proponemos de nuevo un algoritmo genético combinado con búsqueda tabú,

y mediante unos experimentos previos comprobamos que la opción más eficiente es utilizar la estructura de vecindad N_1^S y considerar todos los caminos críticos de la instancia. A continuación mostramos resultados sobre varios benchmarks convencionales del JSP clásico. Comparamos nuestro algoritmo con otros métodos representativos del estado del arte: el algoritmo A^* con un método de poda por dominancia (A^*-PD) y el algoritmo A^* con un método de ponderación heurística (A^*-DW), ambos explicados con detalle por Sierra en [125]. También consideramos el algoritmo de búsqueda local de tipo large step random walk (LSRW) propuesto por Kreipl en [79]. Los resultados indican que el algoritmo propuesto aquí supera a los otros métodos en las instancias de mayor tamaño, obteniendo también resultados muy competitivos en instancias pequeñas. Además, también ofrecemos resultados en benchmarks del SDST-JSP y nos comparamos de nuevo a los obtenidos por el ILOG CPLEX CP Optimizer, debido a que no conocemos trabajos que traten de minimizar esta función objetivo en este problema en particular. Los resultados de nuestro algoritmo mejoran a los obtenidos por el CP Optimizer.

En resumen, la correcta combinación de los métodos y estructuras propuestos en esta tesis ha obtenido resultados excelentes en todas las funciones objetivo estudiadas, superando en muchos casos a los mejores resultados que se pueden encontrar en la literatura. En nuestra opinión, los puntos fuertes del método son la eficacia del algoritmo de búsqueda tabú, su combinación con el algoritmo genético, las estructuras de vecindad propuestas y el método de estimación de vecinos. La estructura N^S, incluso considerando un gran número de vecinos, es muy eficiente gracias a que es capaz de descartar muchos de ellos gracias a las condiciones de factibilidad y de no mejora propuestas para ella.

Sin embargo, para las funciones objetivo de tipo suma, aunque N^S ha obtenido también muy buenos resultados, en condiciones de igualdad de tiempos de ejecución N_1^S proporciona resultados ligeramente mejores. Esto es debido a que en este tipo de funciones existen en general muchos más caminos críticos en una planificación, y además el algoritmo de estimación es más costoso computacionalmente. Por ello, la diferencia en tiempo de ejecución entre una vecindad más simple y una más compleja es mucho más acusada que en una función objetivo de tipo bottleneck, y eso permite a la vecindad más simple utilizar parámetros de población, generaciones e iteraciones de búsqueda tabú bastante más elevados en el mismo tiempo.

12.2. Líneas futuras de investigación

El trabajo desarrollado durante esta tesis deja abiertas diversas líneas de investigación, y en esta sección comentaremos algunas de las principales.

A lo largo de esta tesis hemos tratado cuatro funciones objetivo por separado, pero sería interesante tratar de realizar una optimización multiobjetivo de varias funciones simultáneamente. Este tipo de optimizaciones genera una serie de nuevos problemas y dificultades, pero sin embargo son útiles en problemas reales porque muchas veces la calidad de una planificación depende de varios criterios, y no sólo de uno. La principal diferencia entre optimizaciones monoobjetivo y multiobjetivo es que en el segundo caso lo habitual es que el algoritmo utilizado, en lugar de devolvernos una única planificación, nos devuelva el *frente pareto*, que es un conjunto de planificaciones no dominadas. Posteriormente un operario humano podría revisar este conjunto de soluciones y elegir la que mejor se adapte a sus necesidades en cada momento.

Por otra parte, durante los últimos años se ha observado que la combinación de la búsqueda tabú con otras metaheurísticas ofrece por lo general resultados muy competitivos. Hemos visto algunos ejemplos de este tipo de híbridos en la sección 6.4.2, en los que por ejemplo se combinaba la búsqueda tabú con algoritmos de búsqueda dispersa, shifting bottleneck o colonias de hormigas. También hemos comentado que Beck et al. en [26] combinan el algoritmo de búsqueda tabú i-TSAB propuesto por Nowicki y Smutnicki en [106] con el método *Solution Guided Search* (SGS) propuesto por Beck en [25], y esta metaheurística híbrida es capaz de superar a los mejores métodos del estado del arte en la minimización del makespan en el JSP. Podemos ver otro ejemplo en el trabajo de Streeter y Smith en [137], en donde proponen una combinación de un algoritmo de ramificación y poda con búsqueda local para resolver el JSP. Nosotros nos hemos planteado combinar la búsqueda tabú con métodos de búsqueda heurística en espacios de estados, ya que son las dos especialidades de este grupo de investigación. La búsqueda tabú se podría utilizar como una forma muy eficiente de calcular cotas superiores para un determinado estado, y además su coste computacional no necesitaría ser muy elevado, debido a que cada estado representa un subproblema del problema original y por tanto algunos de los arcos del grafo solución están ya fijados. La idea sería aplicar una combinación de este tipo a la resolución de problemas de scheduling, por ejemplo el JSP o alguna de sus variantes, y esperamos que los resultados sean también muy competitivos.

Otra ampliación interesante es tratar problemas de scheduling más complejos y más

cercanos a los entornos reales de producción. Por ejemplo una variante que tiene bastante interés actualmente es el job shop con operarios. La diferencia de este problema respecto al job shop tradicional es que en este caso las máquinas deben ser manejadas por operarios, y el número de operarios es menor que el número de máquinas, por lo que no todas pueden estar procesando tareas al mismo tiempo. Este problema es de gran interés en aplicaciones reales y su resolución genera nuevas dificultades, aunque no ha sido muy tratado en la literatura. Podemos citar el reciente trabajo de Agnetis et al. en [4], en el que tratan de minimizar el makespan mediante heurísticos de programación dinámica, y el trabajo de Mencía et al. en [91], en el que utilizan un algoritmo genético y son capaces de mejorar los resultados presentados en [4].

Otra variante más cercana a los entornos reales de producción es el job shop flexible con tiempos de setup (SDST-FJSP). Recordemos que en el SDST-JSP cada tarea debe ser procesada por una máquina en concreto, y cada una de las tareas de un mismo trabajo debe utilizar una máquina diferente. La principal diferencia del SDST-FJSP respecto al SDST-JSP es que para cada una de las tareas se define un subconjunto de máquinas en las que puede ser procesada, y además puede ocurrir que una misma máquina procese varias tareas del mismo trabajo. Existe poca literatura sobre esta variante del job shop, y sin embargo es de gran interés en aplicaciones reales. Recientemente, Oddi et al. en [108] tratan la minimización del makespan para este problema, utilizando una avanzada metaheurística denominada IFS (Iterative Flattening Search).

Por otra parte, en esta tesis hemos considerado únicamente funciones objetivo regulares, es decir, que su valor es no decreciente si aumenta el tiempo de fin de los trabajos. Una línea de investigación muy interesante es el estudio de funciones objetivo no regulares, por ejemplo las medidas de estabilidad o robustez. En los entornos de producción reales ocurren a menudo incidencias que llegan a hacer poco útil la solución obtenida inicialmente. Por ejemplo, puede ocurrir que se altere la duración de una tarea, que una máquina se estropee o que aparezca alguna tarea nueva. Aunque no existe consenso sobre la definición exacta del concepto de robustez, la idea es obtener una planificación que sea capaz de mantener su validez o calidad ante una modificación en el problema original, o como mucho que se requieran cambios mínimos en dicha planificación si ocurre alguna incidencia. Éste es un tema de gran interés en aplicaciones prácticas, y es cada vez más considerado en la literatura. Podemos citar como trabajos recientes el de Roy en [122] o los de Van de Vonder et al. en [144] y [145].

12.3. Publicaciones

A continuación presentamos, en orden cronológico, las publicaciones relacionadas con la investigación realizada a lo largo de esta tesis doctoral.

- González, M.A., Vela, C.R. y Varela, R. Scheduling with memetic algorithms over the spaces of semi-active and active schedules. *Lecture Notes in Artificial Intelligence*, volumen 4029, páginas 370-379. 2006.

- González, M.A., Sierra, M., Vela, C.R. y Varela, R. Genetic algorithms hybridized with greedy algorithms and local search over the spaces of active and semi-active schedules. *Current Topics in Artificial Intelligence. 11th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2005. Revised Selected Papers.* LNCS 4177, páginas 231-240. 2006.

- González, M.A., Sierra, M., Vela, C.R., Varela, R. y Puente, J. Combining metaheuristics for the job shop scheduling problem with sequence dependent setup times. *Proceedings of the First International Conference on Software and Data Techonoliges, ICSOFT'2006*, páginas 211-220. 2006.

- González, M.A., Vela, C.R., Sierra, M., González Rodríguez, I. y Varela, R. Comparing schedule generation schemes in memetic algorithms for the job shop scheduling problem with sequence dependent setup times. *Proceedings of MICAI'2006*, páginas 472-482. 2006.

- González, M.A., Sierra, M., Varela, R., Vela, C.R. y Puente, J. Combining metaheuristics for the job shop scheduling problem with sequence dependent setup times. *Software and Data Technologies (Revised and Selected Papers ICSOFT 2006)*, páginas 348-360, CCIS 10, Springer, Germany. 2008.

- González, M.A., Vela, C.R. y Varela, R. A New Hybrid Genetic Algorithm for the Job Shop Scheduling Problem with Setup Times. *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008)*, páginas 116-123, AAAI Press, Sidney. 2008.

- González, M.A., Vela, C.R. y Varela, R. Genetic Algorithm combined with Tabu Search for the Job shop Scheduling Problem with Setup Times. *Third International Work-*

Conference on the Interplay between Natural and Artificial Computation, IWINAC, páginas 265-274, LNCS-5601, Springer. 2009.

- González, M.A., Vela, C.R. y Varela, R. A Tabu Search Algorithm to Minimize Lateness in Scheduling Problems with Setup Times. *Proceedings of CAEPIA 2009*, páginas 115-124. 2009.

- González, M.A., Vela, C.R. y Varela, R. Tabu Search and Genetic Algorithm for Scheduling with Total Flow Time Minimization. *Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems (COPLAS 2010)*, páginas 33-41. 2010.

- Vela, C.R., Varela, R. y González, M.A. Local Search and Genetic Algorithm for the Job Shop Scheduling Problem with Sequence Dependent Setup Times. *Journal of Heuristics*, volume 16, páginas 139-165. DOI 10.1007/s10732-008-9094-y. 2010.

- González, M.A., Vela, C.R. y Varela, R. A Competent Memetic Algorithm for Complex Scheduling. *Aceptado en la revista Natural Computing, pero todavía no publicado.*

- González, M.A., Vela, C.R., González Rodríguez, I. y Varela, R. Lateness Minimization with Tabu Search for Job Shop Scheduling Problem with Sequence Dependent Setup Times. *En proceso de revisión en la revista European Journal of Operational Research.*

- González, M.A., Vela, C.R. y Varela, R. Weighted Tardiness Minimization in Job Shops with Setup Times by Hybrid Genetic Algorithm. *En proceso de revisión en el CAEPIA 2011 Workshop on New Frontiers in Constraint Satisfaction Techniques for Planning and Scheduling Problems.*

Bibliografía

[1] E. Aarts and J. Korst. *Simulated Annealing and Boltzmann Machines*. Wiley, 1989.

[2] E. Aarts, P. Van Laarhoven, J. Lenstra, and N. Ulder. A computational study of local search algorithms for job shop scheduling. *ORSA Journal on Computing*, 6:118–125, 1994.

[3] J. Adams, E. Balas, and D. Zawack. The shifting bottleneck procedure for job shop scheduling. *Managament Science*, 34:391–401, 1988.

[4] A. Agnetis, M. Flamini, G.Ñicosia, and A. Pacifici. A job-shop problem with one additional resource type. *Journal of Scheduling*, DOI 10.1007/s10951-010-0162-4, 2010.

[5] A. Allahverdi, J. Gupta, and T. Aldowaisan. A review of scheduling research involving setup considerations. *Omega*, 27:219–239, 1999.

[6] A. Allahverdi, C. Ng, T. Cheng, and M. Kovalyov. A survey of scheduling problems with setup times or costs. *European Journal of Operational Research*, 187:985–1032, 2008.

[7] K. Amirthagadeswaran and V. Arunachalam. Improved solutions for job shop scheduling problems through genetic algorithm with a different method of schedule deduction. *International Journal on Advanced Manufacturing Technologies*, 28:532–540, 2006.

[8] K. Amirthagadeswaran and V. Arunachalam. Enhancement of performance of genetic algorithm for job shop scheduling problems through inversion operator. *International Journal on Advanced Manufacturing Technologies*, 32:780–786, 2007.

[9] D. Applegate and W. Cook. A computational study of the job-shop scheduling problem. *ORSA Journal of Computing*, 3:149–156, 1991.

[10] V. Armentano and O. Basi de Araujo. Grasp with memory-based mechanisms for minimizing total tardiness in single machine scheduling with setup times. *Journal of Heuristics*, 12(6):427–446, 2006.

[11] V. Armentano and M. Filho. Minimizing total tardiness in parallel machine scheduling with setup times: An adaptive memory-based grasp approach. *European Journal of Operational Research*, 183:100–114, 2007.

[12] V. Armentano and C. Scrich. Tabu search for minimizing total tardiness in a job shop. *International Journal of Production Economics*, 63:131–140, 2000.

[13] C. Artigues, S. Belmokhtar, and D. Feillet. *A new exact solution algorithm for the job shop problem with sequence-dependent setup times*, pages 37–49. 1st international conference on integration of AI and OR techniques in constraint programming for combinatorial optimization problems. LNCS 3011. Springer-Verlag, 2004.

[14] C. Artigues, F. Buscaylet, and D. Feillet. *Lower and upper bound for the job shop scheduling problem with sequence-dependent setup times.* In proceedings of the second Multidisciplinary International Conference on Scheduling: Theory and Applications (MISTA). 2005.

[15] C. Artigues and D. Feillet. A branch and bound method for the job-shop problem with sequence-dependent setup times. *Annals of Operations Research*, 159(1):135–159, 2008.

[16] C. Artigues and P. Lopez. Extending Giffler-Thompson algorithm to generate active schedules for job-shops with sequence-dependent setup times. *LIA report 129, University of Avignon*, 2000.

[17] C. Artigues, P. Lopez, and P. Ayache. Schedule generation schemes for the job shop problem with sequence-dependent setup times: Dominance properties and computational analysis. *Annals of Operations Research*, 138:21–52, 2005.

[18] E. Aydin and T. Fogarty. A distributed evolutionary simulated annealing algorithm for combinatorial optimization problems. *Journal of Heuristics*, 10(3):269–292, 2004.

[19] E. Balas, N. Simonetti, and A. Vazacopoulos. Job shop scheduling with setup times, deadlines and precedence constraints. In *Proceedings of MISTA*, 2005.

[20] E. Balas, N. Simonetti, and A. Vazacopoulos. Job shop scheduling with setup times, deadlines and precedence constraints. *Journal of Scheduling*, 11:253–262, 2008.

[21] E. Balas and A. Vazacopoulos. Guided local search with shifting bottleneck for job shop scheduling. *Management Science*, 44 (2):262–275, 1998.

[22] P. Baptiste and C. Le Pape. Scheduling a single machine to minimize a regular objective function under setup constraints. *Discrete Optimization*, 2:83–99, 2005.

[23] P. Baptiste, C. Le Pape, and W.Ñuijten, editors. *Constraint-Based Scheduling*. Kluwer Academic Publishers, Norwell, MA, USA, 2001.

[24] J. Beck and M. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1):31–81, 2000.

[25] J. C. Beck. Solution-guided multi-point constructive search for job shop scheduling. *Journal of Artificial Intelligence Research*, 29:49–77, 2007.

[26] J. C. Beck, T. Feng, and J.-P. Watson. Combining constraint programming and local search for job-shop scheduling. *Informs Journal on Computing*, DOI: 10.1287/ijoc.1100.0388, 2010.

[27] C. Bierwirth. A generalized permutation approach to jobshop scheduling with genetic algorithms. *OR Spectrum*, 17:87–92, 1995.

[28] U. Bilge, F. Kiraç, M. Kurtulan, and P. Pekgün. A tabu search algorithm for parallel machine total tardiness problem. *Computers & Operations Research*, 31:397–414, 2004.

[29] J. Blazewicz, K. Ecker, E. Pesch, G. Schmidt, and J. Werglarz. *Scheduling Computer and Manufacturing Processes*. Springer, 1996.

[30] P. Brucker. *Scheduling Algorithms*. Springer, 4th edition, 2004.

[31] P. Brucker, B. Jurisch, and B. Sievers. A branch and bound algorithm for the job-shop scheduling problem. *Discrete Applied Mathematics*, 49:107–127, 1994.

[32] P. Brucker and S. Knust. *Complex Scheduling*. Springer, 2006.

[33] P. Brucker and O. Thiele. A branch and bound method for the general-job shop problem with sequence-dependent setup times. *Operations Research Spektrum*, 18:145–161, 1996.

[34] M. Candido, S. Khator, and R. Barcias. A genetic algorithm based procedure for more realistic job shop scheduling problems. *International Journal of Production Research*, 36 (12):3437–3457, 1998.

[35] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11:42–47, 1982.

[36] J. Carlier and E. Pinson. An algorithm for solving the job-shop problem. *Management Science*, 35(2):164–176, 1989.

[37] J. Carlier and E. Pinson. Adjustment of heads and tails for the job-shop problem. *European Journal of Operational Research*, 78:146–161, 1994.

[38] D. Carroll. *Heuristic Sequencing of Jobs with Single and Multiple Components*. PhD thesis, Sloan School of Management, MIT, 1965.

[39] R. Cheng, M. Gen, and Y. Tsujimura. A tutorial survey of job-shop scheduling problems using genetic algorithms-I: representation. *Computers and Industrial Engineering*, 30(4):986–995, 1996.

[40] W. Cheung and H. Zhou. Using genetic algorithms and heuristics for job shop scheduling with sequence-dependent setup times. *Annals of Operations Research*, 107:65–81, 2001.

[41] I. Choi and D. Choi. A local search algorithm for jobshop scheduling problems with alternative operations and sequence-dependent setups. *European Journal of Operational Research*, 90(3):252–268, 2002.

[42] I. Choi and O. Korkmaz. Job shop scheduling with separable sequence-dependent setup times. *Annals of Operations Research*, 70:155–170, 1997.

[43] H. Crauwels, C. Potts, and L. Van Wassenhove. Local search heuristics for single machine scheduling with batch set-up times to minimize total weighted completion time. *Annals of Operations Research*, 70:261–279, 1997.

[44] R. Dawkins. *The selfish gene*. Oxford University Press, 1976.

[45] K. DeBontridder. Minimizing total weighted tardiness in a generalized job shop. *Journal of Scheduling*, 8:479–496, 2005.

[46] M. Dell' Amico and M. Trubian. Applying tabu search to the job-shop scheduling problem. *Annals of Operational Research*, 41:231–252, 1993.

[47] E. Demirkol, S. Mehta, and R. Uzsoy. A computational study of shifting bottleneck procedures for shop scheduling problems. *Journal of Heuristics*, 3 (2):111–137, 1997.

[48] U. Dorndorf, E. Pesch, and T. Phan-Huy. Constraint propagation techniques for the disjunctive scheduling problem. *Artificial Intelligence*, 122:189–240, 2000.

[49] I. Essafi, Y. Mati, and S. Dauzère-Pérès. A genetic local search algorithm for minimizing total weighted tardiness in the job-shop scheduling problem. *Computers and Operations Research*, 35:2599–2616, 2008.

[50] V. Eswaramurthy and A. Tamilarasi. Hybridizing tabu search with ant colony optimization for solving job shop scheduling problems. *International Journal on Advanced Manufacturing Technologies*, 40:1004–1015, 2009.

[51] H. Fisher and G. L. Thompson. Probabilistic learning combinations of lcoal job-shop scheduling rules. In J. F. Muth and G. L. Thompson, editors, *Industrial Scheduling*, pages 225–251. Prentice Hall, 1963.

[52] F. Focacci, P. Laborie, and W. Nuijten. Solving scheduling problems with setup times and alternative resources. In *Proceedings of Fifth International Conference on Artificial Intelligence Planning and Scheduling*, pages 92–101, 2000.

[53] J. Fowler, S. Horng, and J. Cochran. A hybridized genetic algorithm to solve parallel machine scheduling problems with sequence dependent setups. *International Journal of Industrial Engineering*, 10:232–243, 2003.

[54] M. Garey, D. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976.

[55] B. Giffler and G. L. Thompson. Algorithms for solving production scheduling problems. *Operations Research*, 8:487–503, 1960.

[56] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13:533–549, 1986.

[57] F. Glover. Tabu search–part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

[58] F. Glover. Tabu search–part II. *ORSA Journal on Computing*, 2(1):4–32, 1989.

[59] F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[60] D. Goldberg. *Genetic algorithms in search. Optimization and machine learning*. Addison-Wesley, 1985.

[61] M. A. González, M. Sierra, C. R. Vela, and R. Varela. Genetic algorithms hybridized with greedy algorithms and local search over the spaces of active and semi-active schedules. *Current Topics in Artificial Intelligence. 11th Conference of the Spanish Association for Artificial Intelligence, CAEPIA 2005. Revised Selected Papers. LNCS 4177*, pages 231–240, 2006.

[62] M. A. González, C. Vela, and R. Varela. A new hybrid genetic algorithm for the job shop scheduling problem with setup times. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS-2008)*, pages 116–123, Sidney, 2008. AAAI Press.

[63] M. A. González, C. R. Vela, and R. Varela. Scheduling with memetic algorithms over the spaces of semi-active and active schedules. *Lecture Notes in Artificial Intelligence*, 4029:370–379, 2006.

[64] J. Grabowsky and M. Wodecki. *A very fast tabu search algorithm for job shop problem.* Operations Research/Computer Science Interfaces Series.

[65] R. Graham, E. Lawler, J. Lenstra, and A. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[66] P. Hansen and N. Mladenovic. A tutorial on variable neighborhood search. *Les Cahiers du GERAD*, 46, 2003.

[67] J. Hao, R. Dorne, and P. Galinier. Tabu search for frequency assignment in mobile radio networks. *Journal of Heuristics*, 4(1):47–62, 1998.

[68] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* The University of Michigan Press, 1975.

[69] O. Holthaus and C. Rajendram. Efficient dispatching rules for scheduling in a job shop. *International Journal of Production Economics*, 48(1):87–105, 1997.

[70] K. Huang and C. Liao. Ant colony optimization combined with taboo search for the job shop scheduling problem. *Computers and Operations Research*, 35:1030–1046, 2008.

[71] L. Ingolotti Hetter. *Modelos y Métodos para la Optimización y Eficiencia de la Programación de Horarios Ferroviarios*. PhD thesis, Universidad Politécnica de Valencia, 2007.

[72] P. Ives and M. Lambrecht. Extending the shifting bottleneck procedure to real-life applications. *European Journal of Operational Research*, 90 (3):252–268, 1996.

[73] A. Jain, B. Rangaswarny, and S. Meeran. New and "stronger" job-shop neighbourhoods: A focus on the method of Nowicki and Smutnicki (1996). *Journal of Heuristics*, 6 (4):457–480, 2000.

[74] A. Kahn. Topological sorting in large networks. *Communications of the ACM*, 5:558–562, 1962.

[75] S. Kim and P. Bobrowski. Impact of sequence-dependent setup time on job shop scheduling performance. *International Journal of Production Research*, 32 (7):1503–1520, 1994.

[76] S. KirkPatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[77] C. Koulamas and G. Kyparisis. Single-machine scheduling problems with sequence-depenent setup times. *European Journal of Operational Research*, 187:1045–1049, 2008.

[78] N. Krasnogor and J. Smith. A tutorial for competent memetic algorithms: model, taxonomy, and design issues. *IEEE Transactions on Evolutionary Computation*, 9(5):474–488, 2005.

[79] S. Kreipl. A large step rendom walk for minimizing total weighted tardiness in a job shop. *Journal of Scheduling*, 3:125–138, 2000.

[80] P. Laborie. Algorithms for propagating resource constraints in ai planning and scheduling: Existing approaches and new results. *Artificial Intelligence*, 143(2):151–188, 2003.

[81] M. Laguna, J. Barnes, and F. Glover. Tabu search methods for a single machine scheduling problem. *Journal of Intelligent Manufacturing*, 2:63–74, 1991.

[82] S. Lawrence. Resource constrained project scheduling: an experimental investigation of heuristic scheduling techniques (supplement). Technical report, Graduate School of Industrial Administration, Carnegie Mellon University, 1984.

[83] Y. Lee and M. Pinedo. Scheduling jobs on parallel machines with sequence-dependent setup times. *European Journal of Operational Research*, 100:464–474, 1997.

[84] T. Liu, J. Tsai, and J. Chou. Improved genetic algorithm for the job-shop scheduling problem. *International Journal on Advanced Manufacturing Technologies*, 27:1021–1029, 2006.

[85] O. Martin, S. Otto, and E. Felten. Large step markov chains for the TSP incorporating local search heuristics. *Operations Research Letters*, 11:219–224, 1992.

[86] Y. Mati, S. Dauzere-Peres, and C. Lahlou. A general approach for optimizing regular criteria in the job-shop scheduling problem. *European Journal of Operational Research*, DOI: 10.1016/j.ejor.2011.01.046, 2011.

[87] H. Matsuo, C. Suh, and R. Sullivan. A controlled search simulated annealing method for the general jobshop scheduling problem. Working paper 03-44-88, Graduate School of Business, University of Texas, 1988.

[88] D. Mattfeld. *Evolutionary Search and the Job Shop Investigations on Genetic Algorithms for Production Scheduling*. Springer-Verlag, 1995.

[89] D. Mattfeld and C. Bierwirth. An efficient genetic algorithm for job shop scheduling with tardiness objectives. *European Journal of Operational Research*, 155:616–630, 2004.

[90] S. Mehta and R. Uzsoy. Predictable scheduling of a job shop subject to breakdowns. *IEEE Transactions on Robotics and Automation*, 14(3):365–378, 1998.

[91] R. Mencía, M. Sierra, C. Mencía, and R. Varela. Genetic algorithm for job-shop scheduling with operators. In *Fourth International Work-Conference on the Interplay between Natural and Artificial Computation, IWINAC*, pages 305–314. LNCS-6687, 2011.

[92] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculation by fast computing machines. *Journal of Chemistry Physics*, 21:1087–1091, 1953.

[93] Z. Michalewicz. *Genetic algorithms + Data structures = Evolution program*. Springer-Verlag, second, extended edition, 1994.

[94] D. Miller, H. Chen, J. Matson, and Q. Liu. A hybrid genetic algorithm for the single machine scheduling problem. *Journal of Heuristics*, 5(4):437–454, 1999.

[95] L. Mönch, R. Schabacker, D. Pabst, and J. Fowler. Genetic algorithm-based subproblem solution procedures for a modified shifting bottleneck heuristic for complex job shops. *European Journal of Operational Research*, 177 (3):2100–2118, 2007.

[96] P. Moscato. On evolution, search, optimization, genetic algorithms and martial arts: Towards memetic algorithms. *Technical Report 826, Caltech Concurrent Computation Program Report, Caltech, Pasadena, California*, 1989.

[97] B. Murovec and P. Suhel. A repairing technique for the local search of the job-shop problem. *European Journal of Operational Research*, 153:220–238, 2004.

[98] B. Naderi, S. Fatemi Ghomi, and M. Aminnayeri. A high performing metaheuristic for job shop scheduling with sequence-dependent setup times. *Applied Soft Computing*, 10:703–710, 2010.

[99] B. Naderi, M. Zandieh, and S. Fatemi Ghomi. Scheduling job shops with sequence dependent setup times. *International Journal of Production Research*, 47:5959–5976, 2009.

[100] B. Naderi, M. Zandieh, and S. Fatemi Ghomi. A study on integrating sequence dependent setup time flexible flow lines and preventive maintenance scheduling. *Journal on Intelligent Manufacturing*, 20:683–694, 2009.

[101] B. Naderi, M. Zandieh, A. Khaleghi Ghoshe Balagh, and V. Roshanaei. An improved simulated annealing for hybrid flowshops with sequence-dependent setup and transportation times to minimize total completion time and total tardiness. *Expert Systems with Applications*, 36:9625–9633, 2009.

[102] M. Nasiri and F. Kianfar. A hibryd scatter search for the partial job shop scheduling problem. *International Journal on Advanced Manufacturing Technologies*, DOI 10.1007/s00170-010-2792-2, 2010.

[103] N.Ñilsson. *Principles of Artificial Intelligence*. Tioga, Palo Alto, CA, 1980.

[104] A. Noivo and H. Ramalhinho-Lourenço. Solving two production scheduling problems with sequence-dependent set-up times. *Technical Report 138, Department of Economic and Business, Universitat Pompeu Fabra, Barcelona*, 1998.

[105] E. Nowicki and C. Smutnicki. A fast taboo search algorithm for the job shop scheduling problem. *Management Science*, 42:797–813, 1996.

[106] E. Nowicki and C. Smutnicki. An advanced tabu search algorithm for the job shop problem. *Journal of Scheduling*, 8:145–159, 2005.

[107] A. Oddi, R. Rasconi, A. Cesta, and S. Smith. Iterative-sampling search for job shop scheduling with setup times. In *COPLAS 2009 Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, pages 27–33, 2009.

[108] A. Oddi, R. Rasconi, A. Cesta, and S. Smith. Applying iterative flattening search to the job shop scheduling problem with alternative resources and sequence dependent setup times. In *COPLAS 2011 Proceedings of the Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, 2011.

[109] I. Ovacik and R. Uzsoy. Exploiting shop floor status information to schedule complex job shops. *Journal of Manufacturing Systems*, 13(2):73–84, 1994.

[110] I. Ovacik and R. Uzsoy. *Decomposition methods for complex factory scheduling problems*. Kluwer Academic, 1997.

[111] J. Pearl. *Heuristics: Intelligent Search strategies for Computer Problem Solving*. Addison-Wesley, 1984.

[112] F. Pezzella and E. Merelli. A tabu search method guided by shifting bottleneck for job shop scheduling problem. *European Journal of Operational Research*, 120:297–310, 2000.

[113] D. Philip. *Scheduling reentrant flexible job shops with sequence dependent setup times*. PhD thesis, University of Montana, 2005.

[114] M. Pinedo. *Planning and Scheduling in Manufacturing and Services*. Springer, 2005.

[115] M. Pinedo. *Scheduling. Theory, Algorithms, and Systems*. Springer, third edition, 2008.

[116] S. Ponnambalam, N. Jawahar, and B. Kumar. Estimation of optimum genetic control parameters for job shop scheduling. *International Journal on Advanced Manufacturing Technologies*, 19:224–234, 2002.

[117] J. Puente, C. Vela, C. Prieto, and R. Varela. *Artificial Neural Nets Problem Solving Methods (IWANN 2003)*, volume LNCS 2687, chapter Hybridizing a Genetic Algorithm with Local Search and Heuristic Seeding, pages 329–336. Springer, 2003.

[118] R. Rardin and R. Uzsoy. Experimental evaluation of heuristic optimization algorithms: A tutorial. *Journal of Heuristics*, 7:261–304, 2001.

[119] R. Ríos-Mercado and J. Bard. An enhanced TSP-based heuristic for makespan minimization in a flow shop with setup times. *Journal of Heuristics*, 5(1):53–70, 1999.

[120] E. Rodriguez-Tello, J. Hao, and Torres-Jimenez. An effective two-stage simulated annealing algorithm for the minimum arrangement problem. *Computer and Operation Research*, 35(10):3331–3346, 2008.

[121] V. Roshanaei, B. Naderi, F. Jolai, and M. Khalili. A variable neighborhood search for job shop scheduling with set-up times to minimize makespan. *Future Generation Computer Systems*, 25:654–661, 2009.

[122] B. Roy. Robustness in operational research and decision aiding: A multi-faceted issue. *European Journal of Operational Research*, 200:629–638, 2010.

[123] B. Roy and B. Sussmann. Les problèmes d'ordonnancement avec constraintes disjonctives. Note d.s. no. 9 bis, d6c, SEMA, Matrouge, Paris, 1964.

[124] F. Serifoglu and G. Ulusoy. Parallel machine scheduling with earliness and tardiness penalties. *Computers & Operations Research*, 26:773–787, 1999.

[125] M. Sierra. *Mejora de Algoritmos de Búsqueda Heurística mediante Poda por Dominancia. Aplicación a Problemas de Scheduling.* PhD thesis, Universidad de Oviedo, 2009.

[126] M. Sierra, C. Mencía, and R. Varela. Weighting disjunctive heuristics for scheduling problems with summation cost functions. *Proceedings of Workshop on Planning, Scheduling and Constraint Satisfaction, CAEPIA*, 2009.

[127] M. Sierra and R. Varela. Pruning by dominance in best-first search. In *Proceedings of CAEPIA'2007*, volume 2, pages 289–298, 2007.

[128] M. Sierra and R. Varela. A new admissible heuristic for the job shop scheduling problem with total flow time. *ICAPS-2008. Workshop on Constraint Satisfaction Techniques for Planning and Scheduling. Sidney*, 2008.

[129] M. Sierra and R. Varela. Pruning by dominance in best-first search for the job shop scheduling problem with total flow time. *Journal of Intelligent Manufacturing, DOI 10.1007/s10845-008-0167-4*, 1:1–2, 2008.

[130] M. Sierra and R. Varela. Best-first search and pruning by dominance for the job shop scheduling problem with total flow time. *Journal of Intelligent Manufacturing*, 21(1):111–119, 2010.

[131] M. Singer. Decomposition methods for large job shops. *Computers and Operations Research*, 28:193–207, 2001.

[132] M. Singer and M. Pinedo. A computational study of branch and bound techniques for minimizing the total weighted tardiness in job shops. *IIE Transactions*, 30:109–118, 1998.

[133] M. Singer and M. Pinedo. A shifting bottleneck heuristic for minimizing the total weighted tardiness in a job shop. *Naval Research Logistics*, 46(1):1–17, 1999.

[134] S. Smith and C. Cheng. Slack-based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence*, 1993.

[135] G. Stecco and J.-F. Cordeau. A tabu search heuristic for a sequence-dependent and time-dependent scheduling problem on a single machine. *Journal of Scheduling*, 12:3–16, 2009.

[136] R. Storer, S. Wu, and R. Vaccari. New search spaces for sequencing problems with application to job shop scheduling. *Management Science*, 38(10):1495–1509, 1992.

[137] M. Streeter and S. Smith. Exploiting the power of local search in a branch and bound algorithm for job shop scheduling. In *Proceedings of the Sixteenth International Conference on Automated Planning and Scheduling*, pages 324–332, 2006.

[138] C. Suh. *Controlled search simulated annealing for job shop scheduling.* PhD thesis, University of Texas, 1988.

[139] X. Sun and J. Noble. An approach to job shop scheduling with sequence-dependent setups. *Journal of Manufacturing Systems*, 18(6):416–430, 1999.

[140] E. Taillard. Parallel taboo search techniques for the job shop scheduling problem. *ORSA Journal on Computing*, 6:108–117, 1993.

[141] R. Uzsoy and J. Velásquez. Heuristics for minimizing maximum lateness on a single machine with family-dependent set-up times. *Computers and Operations Research*, 35:2018–2033, 2008.

[142] R. Vaessens, E. Aarts, and J. Lenstra. Job shop scheduling by local search. *INFORMS Journal on Computing*, 3:302–317, 1996.

[143] E. Vallada and R. Ruiz. A genetic algorithm for the unrelated parallel machine scheduling problem with sequence dependent setup times. *European Journal of Operational Research*, 2011.

[144] S. Van de Vonder, E. Demeulemeester, and W. Herroelen. A classification of predictive-reactive project scheduling procedures. *Journal of Scheduling*, 10:195–207, 2007.

[145] S. Van de Vonder, E. Demeulemeester, and W. Herroelen. Proactive heuristic procedures for robust project scheduling: An experimental analysis. *European Journal of Operational Research*, 189:723–733, 2008.

[146] P. Van Hentenryck and L. Michel. Scheduling abstractions for local search. *Lecture Notes in Computer Science*, 3011:319–334, 2004.

[147] P. Van Laarhoven and E. Aarts. *Simulated Annealing - Theory and Applications*. Kluwer, 1988.

[148] P. Van Laarhoven, E. Aarts, and K. Lenstra. Job shop scheduling by simulated annealing. *Operations Research*, 40:113–125, 1992.

[149] R. Varela, D. Serrano, and M. Sierra. New codification schemas for scheduling with genetic algorithms. *Proceedings of IWINAC 2005. Lecture Notes in Computer Science*, 3562:11–20, 2005.

[150] R. Varela, C. R. Vela, J. Puente, and A. Gómez. A knowledge-based evolutionary strategy for scheduling problems with bottlenecks. *European Journal of Operational Research*, 145:57–71, 2003.

[151] C. R. Vela, R. Varela, and M. A. González. Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times. *Journal of Heuristics*, 16:139–165, 2010.

[152] A. Vepsalainen and T. Morton. Priority rules for job shops with weighted tardiness costs. *Management Science*, 33(8):1035–1047, 1985.

[153] L. Wang and D. Zheng. A modified genetic algorithm for job shop scheduling. *International Journal on Advanced Manufacturing Technologies*, 20:72–76, 2002.

[154] Y. Wang, H. Yin, and J. Wang. Genetic algorithm with new encoding scheme for job shop scheduling. *International Journal on Advanced Manufacturing Technologies*, 44:977–984, 2009.

[155] Y. Wang, H. Yin, and J. Wang. Optimization method with large leap steps for job shop scheduling. *International Journal on Advanced Manufacturing Technologies*, 43:1018–1023, 2009.

[156] J. Watson, A. Howe, and L. Whitley. Deconstructing Nowicki and Smutnicki's i-TSAB tabu search algorithm for the job-shop scheduling problem. *Computers and Operations Research*, 33:2623–2644, 2006.

[157] J. Wilbrecht and W. Prescott. The influence of setup times on job shop performance. *Management Science*, 16(4):391–401, 1969.

[158] T. Wu, C. Chang, and S. Chung. A simulated annealing algorithm for manufacturing cell formation problems. *Expert Systems with Applications*, 34(3):1609–1617, 2008.

[159] X. Xu and C. Li. Research on immune genetic algorithm for solving the job-shop scheduling problem. *International Journal on Advanced Manufacturing Technologies*, 34:783–789, 2007.

[160] T. Yamada and R. Nakano. Scheduling by genetic local search with multi-step crossover. In *Proceedings of Fourth International Conference On Parallel Problem Solving from Nature (PPSN IV)*, pages 960–969, 1996.

[161] C. Zhang, P. Li, Z. Guan, and Y. Rao. A tabu search algorithm with a new neighborhood structure for the job shop scheduling problem. *Computers and Operations Research*, 34:3229–3242, 2007.

[162] C. Zhang, P. Li, Y. Rao, and Z. Guan. A very fast TS/SA algorithm for the job shop scheduling problem. *Computers and Operations Research*, 35:282–294, 2008.

[163] C. Zhang, Y. Rao, and P. Li. An effective hybrid genetic algorithm for the job shop scheduling problem. *International Journal on Advanced Manufacturing Technologies*, 39:965–974, 2008.

[164] H. Zhou, W. Cheung, and L. Leung. Minimizing weighted tardiness of job-shop scheduling using a hybrid genetic algorithm. *European Journal of Operational Research*, 194(1):637–649, 2009.

[165] G. Zobolas, C. Tarantilis, and G. Ioannou. A hybrid evolutionary algorithm for the job shop scheduling problem. *Journal of the Operational Research Society*, 60:221–235, 2009.

[166] J. Zoghby, J. Barnes, and J. Hasenbein. Modeling the re-entrant job shop scheduling problem with setup for metaheuristic searches. *European Journal of Operational Research*, 167:336–348, 2005.