Bandwidth Packing: A Tabu Search Approach

Manuel Laguna • Fred Glover
Graduate School of Business and Administration, Campus Box 419, University of Colorado at Boulder, Boulder, Colorado 80309-0419
U.S. West Chair in Systems Science, Graduate School of Business and Administration, Campus Box 419, University of Colorado at Boulder, Boulder, Colorado 80309-0419

The bandwidth packing (BWP) problem is a combinatorially difficult problem arising in the area of telecommunications. The problem consists of assigning calls to paths in a capacitated graph, such that capacities are not violated and the total profit is maximized. In this paper we discuss the development of a tabu search (TS) method for the BWP problem. The method makes use of an efficient implementation of the k-shortest path algorithm, that allows the identification of a controlled set of feasible paths for each call. A tabu search is then performed to find the best path assignment for each call. The TS method developed here incorporates a number of features that have proved useful for obtaining optimal and near optimal solutions to difficult combinatorial problems. We establish the effectiveness of our approach by comparing its performance in speed and solution quality to other specialized heuristics and to a standard optimization package applied to a 0-1 integer programming formulation of the problem.
(*Tabu Search; k-shortest Paths; Call Routing*)

1. Introduction

A recurrent problem in today's service and manufacturing industries arises where demand requirements exceed the capacities of associated installations for storage and distribution. If the capacity is considered fixed (i.e., no expansion is scheduled in the short term), the problem is then to select jobs or activities that should be performed in order to maximize some measure of effectiveness (e.g., profit or customer satisfaction). In the area of telecommunications, the demand requirements of calls to be routed through a given telecommunication network may exceed the network's installed capacity. The problem is then to select a subset of calls and assign them to feasible paths, usually with the objective of maximizing profit. This problem is referred to as the *bandwidth packing* (BWP) problem (Cox et al. 1990). In this paper we address the *static* version of the BWP problem, where demand (over the current time horizon) is given by a table of calls, and bandwidth requirements are fixed and known with certainty. This version differs from the *dynamic* one that treats demand as a stochastic process.

The BWP problem is defined by reference to a graph with capacitated arcs. Each call has a known revenue and a nonnegative bandwidth requirement, and can be assigned only to a node-simple path in the graph. Associated with each link in the graph is a capacity (given in bandwidth) and a unit cost. Since the demand of the calls exceeds the total capacity of the network, regardless of how calls are routed, the problem is to find an optimal assignment of a feasible subset of calls to paths. We use the following notation:

General. n: number of nodes, p: number of links, q: number of calls.

For each link j. b_j: capacity, c_j: unit cost, f_j: flow.

For each call i. r_i: capacity requirements, v_i: revenue, s_i: source node, t_i: terminal node.

The source and terminal nodes for each call are

0025-1909/93/3904/0492$01.25

interchangeable, and hence are designated arbitrarily, since the flow in all links of the network is bidirectional. (That is, in graph terminology, the links represent undirected arcs, or edges.) Letting **A** denote the set of assigned calls, i.e., those calls with an assigned path, the problem may be expressed mathematically as follows:

$$\text{Maximize} \quad \text{Profit} = \sum_{i \in A} v_i - \sum_{j=1}^{p} c_j * f_j$$

$$\text{subject to} \quad f_j \le b_j, \qquad j = 1, \ldots, p,$$

$$\text{where} \quad f_j = \sum_{i \in A_j} r_i$$

$$\text{and} \quad \mathbf{A}_j = \text{set of all calls routed through link } j.$$

The first term in the objective function identifies the total revenue from all assigned calls, and the second term identifies the cost of using links to which the calls are assigned. Only those links with positive flow f_j contribute to the total cost. The inequality constraints ensure that the capacity of each link is not violated. Flow in each link equals the sum of the bandwidth requirements for each call routed through the link.

Our study of the BWP problem focuses on the development of a tabu search (TS) method capable of obtaining high quality solutions with a reasonable amount of computer effort. In preliminary testing we have used a standard optimization package to solve small problem instances of the BWP problem for purposes of comparison. For such small problems, it is possible to generate all profitable paths (i.e., those for which the revenue does not exceed the total cost) for each call, giving rise to a 0-1 integer programming formulation that can be solved (with a somewhat large computational effort) using available optimization software. The starting point of our method, and also the basis for creating inputs for the optimization software, is to generate profitable paths for each call with an efficient implementation of a k-shortest path algorithm, where each path contains no repeated nodes. We first describe the implementation of this algorithm.

2. Finding k-shortest Paths

The search strategy of our TS method is initiated by computing a prespecified maximum number (k_max) of profitable paths for each call. A path π is profitable for call i if

$$v_i - r_i * \sum_{j \in \mathbf{L}_\pi} c_j > 0,$$

where $\mathbf{L}_\pi$ is the set of links in path π. A k-shortest path algorithm, for paths without repeated nodes, may be used to find a set of profitable paths. (In the case of small problems, this set contains all paths that qualify as profitable.) We elected to implement a variant of the k-shortest path algorithm reported by Lawler (1976). Our implementation uses a *penalized* shortest path algorithm as a subprocedure. This allows us to exclude or include specific links by means of penalties. Specifically, let call i be the selected call. Then the temporary cost $\bar{c}_j$ of each link is given by

$$\bar{c}_j = \begin{cases} c_j * r_i & \text{if } b_j \ge r_i, \\ BIG & \text{otherwise} \end{cases} \quad \text{for } j = 1, \ldots, p,$$

where BIG is a "large cost" equal to $(p+1)\max(c_j) r_i$. If the shortest path has a length of at least BIG, then no feasible path exists for the selected call. Before the k-shortest path procedure is called, the temporary costs for all links are assigned as defined above. Once determined, the shortest path is broken into subpaths that exclude certain links by the prescriptions of the k-shortest path procedure. The process then repeats, temporarily giving excluded links a length of BIG, and continuing until no more profitable paths are present or k_max paths are found. In summary, the penalized shortest path algorithm allows us to search for the k-shortest paths for call i in the graph containing all links j such that $b_j \ge r_i$. The shortest path procedure used in our implementation is fully described in Glover (1990).

3. The Tabu Search Procedure

Tabu search is a meta-level procedure for guiding local search methods to overcome local optimality. TS operates by incorporating flexible memory functions to forbid transitions (*moves*) between solutions that reinstate certain attributes of past solutions. Attributes that are not permitted to be reinstated are called *tabu*, and are maintained in short term memory on a list called a *tabu list*. After a specified duration they are removed from the list and are free to be instated again. Effective TS procedures keep a balance between intensification and diversification, that is, between reinforcing attributes associated with good solutions and driving the search into regions not yet visited. To achieve this balance, intermediate and long term memory functions are used in combination with the short term memory. A recent trend in the tabu search field is the use of dynamic tabu lists, which in their most common form use variable list sizes, as a way of managing short term memory. In the method developed here, we have incorporated both a long term memory function and a strategy to change the tabu list size during the course of the search.

3.1. General Description

The method starts by generating k_i profitable paths for each call i ($i = 1, \ldots, q$) up to a maximum of k_max paths. The total cost of routing a call through any of its profitable paths is known (since these costs are calculated when the paths are generated). An initial solution is found by assigning one call at a time to the most profitable path whose current capacity is sufficient to carry the call. The calls are chosen to be assigned to paths in the sequence given by their revenue-to-requirement ratio ρ (i.e., the call with the largest v_i/r_i value is selected first). If no profitable path currently exists for the call under consideration, then the call is assigned to the null path. The starting procedure finishes when all calls have been considered. Figure 1 shows a C-like pseudo code for this starting procedure. Note that the calls are assumed to be ordered in decreasing ρ-ratio, i.e., $i < j$ implies that $v_i/r_i \geq v_j/r_j$.

Our search approach transforms a given solution into another, starting from the initial solution, as follows. We define a move (i, h), for $i = 1, \ldots, q$ and $h = 0, \ldots, k_i$, to be an operation that changes the assignment of call i from its current path g to path h. After a move (i, h) is performed, the short term memory component

Figure 1 **Starting Solution Generator**

```
A ← $\varnothing$
profit ← 0
for ($i$ = 1, . . . , $q$) {
        cost = select__path(A, $i$, path)
        if (path ≠ NULL) {
                A ← ($i$, path)
                profit ← profit + $v_i$ − cost
        }
}
```

introduces a tabu restriction that forbids call i to be assigned to path g for tabu_size iterations (where tabu_size is the current size of the tabu list). (When $h = 0$ the move assigns call i to a null path, or in other words, the call is not in the set **A**.) At each iteration the method performs the best move available and the tabu information is then updated. The definition of "best move" is a function of the search state and the information contained in the different memory structures. In the next subsection we discuss the criteria that determine the selection of a best move as a function of short term and long term memory.

3.2. Frequency vs. Recency

The short term memory function is designed to avoid moves that will cancel the effect of those recently performed. In general, short term memory serves to prevent the duplication of conditions encountered in the recent past. In many situations however, short term memory alone is not sufficient to drive the search to regions where improved solutions can be found. A complementary use of longer term memory is required in which the notion of frequency plays an important role. The idea behind using frequency information is to redirect local choices (i.e., those that relate to a given neighborhood) by exploiting the knowledge of how often the same choices have been made in the past. The use of frequency counts has proved effective in other settings (Laguna and Glover 1991) when activated during search states where no improving move is available from the set of nontabu moves.

The long term memory function embedded in our method consists of q lists of the form freq(i, h) for $i = 1, \ldots, q$ and $h = 0, \ldots, k_i$. The (i, h) element identifies the number of times that call i has been moved to path h during the search. The best move at any iteration is the nontabu move with the largest penalized move value (pmv). If call i is currently assigned to path g, then the penalized move value for a move (i, h) is given by

$$\text{pmv}(i, h) = \text{move\_value}(i, h) - \alpha * \text{freq}(i, h) \quad \text{where}$$

$$\alpha = \begin{cases} 0 & \text{if move\_value}\,(i, h) > 0, \\ 10 & \text{otherwise;} \end{cases}$$

$$\text{move\_value}\,(i, h) = r_i \left(\sum_{j \in \mathbf{L}_g} c_j - \sum_{j \in \mathbf{L}_h} c_j \right).$$

The parameter α in general can be made to depend on the problem class and solution history. Our choice of $\alpha = 10$ was based on preliminary experimentation that suggested this value worked well in the present setting. The calculation of one move value can be performed in $O(1)$ time, since the cost for each profitable path is known in advance (determined during the preprocessing part of our method by means of the k-shortest path algorithm). The procedure that finds a best admissible move at each iteration examines $O(q*k\_\max)$ candidates, and in the worst case, it requires $O(n)$ operations to check for the feasibility of each path. (This is a conservative estimate, since the actual number of links in any profitable path is much less than n.) Our TS procedure employs a simple aspiration level criterion to override tabu status by allowing a tabu move to be executed if it yields a solution better than the incumbent (i.e., the best found so far).

3.3. Dynamic Tabu List Size

The short term memory function of many TS methods employs a single tabu list with fixed size. Selected attributes of moves are recorded on this list and are released after a prespecified number of iterations. Following a theme suggested in (Glover 1989), recent applications of tabu search (e.g., Hertz and de Werra 1990, Taillard 1990, Skorin-Kapov 1991) have shown that there is merit in allowing the size or structure of the tabu list to vary during the course of the search. In (Taillard 1990), for example, the size of the list is changed approximately every 2*tabu_size iterations from its current value to one that is randomly selected between a minimum and maximum allowed size, while the approach of Skorin-Kapov (1991) uses a "moving gap" tabu list that induces a structural change.

In our study we are concerned with testing the use of multiple dynamic lists whose sizes change systematically rather than randomly. Specifically, each call i has its own tabu list size, tabu_size(i). This size is allowed to take on three possible values that are a linear function of the number of profitable paths k_i for each call. For any call i, small (S), medium (M), and large (L) sizes are set equal to k_i, $1.5*k_i$, and $2*k_i$, respectively. The value of tabu_size(i), for $i = 1, \ldots, q$, is initially set to S (i.e., k_i) and then systematically changed every 2*tabu_size(i) iterations following the sequence {S, M, S, L, M, L}.

The sequence is repeated as many times as necessary until the end of the search. Note that this sequence alternately increases and decreases the size of the tabu lists. When a move (i, h) becomes tabu, it keeps its status for a number of iterations that corresponds to the current size of tabu_list(i). In order to correctly enforce this restriction, q tabu_time lists (each of size k_i, for $i = 1, \ldots, q$) are employed. The hth position of the ith list contains the time (iteration number) in which move (i, h) is to be released from its tabu status. This data structure proves to be very convenient when experimenting with fixed and dynamic sizes for single or multiple lists.

4. An IP Formulation

To allow the problem to be treated by existing integer programming software, BWP may be formulated as a 0-1 integer programming problem as follows. Let profit(i, h) be the profit from assigning call i to path h. Then we express the BWP problem as a binary integer program with $p*q$ inequality constraints and a number of variables that grows with the number of profitable paths:

$$\text{Maximize} \quad \text{Total Profit} = \sum_{i=1}^{q} \sum_{h=1}^{k_i} \text{profit}(i, h)*x_{ih}$$

$$\text{subject to} \quad \sum_{h=1}^{k_i} x_{ih} \le 1, \qquad i = 1, \ldots, q,$$

$$\sum_{i=1}^{q} \sum_{h \mid j \in L_h} r_i * x_{ih} \le b_j, \quad j = 1, \ldots, p, \text{ and}$$

$$x_{ih} = \begin{cases} 1 & \text{if call } i \text{ is assigned to path } h, \\ 0 & \text{otherwise.} \end{cases}$$

Although only profitable paths need to be considered in this formulation, the number of such paths increases rapidly with the number of links. Therefore, only small instances of the BWP problem can be solved by means of standard integer programming techniques. In the next section, preliminary computational experiments are performed that use such a standard 0-1 integer programming routine to solve a small instance of the BWP problem as a means of evaluating the performance of the TS procedure.

5. Initial Computational Experiments

Our first set of experiments undertakes to determine the merit of the long term memory function design and the dynamic list structure. A small example problem is used for this purpose, consisting of a network with 10 nodes and 12 capacitated arcs (see Figure 2). The call table consists of 20 calls with individual bandwidth requirements (demand) and revenue given in Table 1. This example can be formulated as a 0-1 integer programming problem with 96 variables and 32 constraints. The optimal solution yields a profit of $4,991 and was found in 10 CPU minutes on a VAX machine using LINDO (Cox et al. 1990). The largest number of profitable paths for any call is 6 (generated for call 18), while the smallest number of such paths is 1 (for call 20). It is possible for some problems to contain calls for which no profitable (or even feasible) paths exist.

We investigated the joint effect of three factors on the quality of the solution: tabu lists, long term memory function, and maximum number of profitable paths. Solution trials of 15,000 iterations each were performed for four tabu size settings (fixed S, M, L, and dynamic D size), with the long term memory function both enabled and disabled, and for five values of *k*_max (2 to 6). The results of these 40 solution trials are summarized in Table 2. The columns indicate the tabu size employed, while the rows show whether or not the long term memory function (LTMF) was employed. Each cell contains the number of times, NOS, that the optimal solution was found out of the five attempts for each of the *k*_max values. (An optimal solution to the example problem can be obtained by setting *k*_max to a value as low as 2.) In addition, each cell shows the average percentage deviation from optimality (PDO) and the average iteration number where the best solution was found (ITB). The average time to complete each solution attempt, shown in Table 3, depends on the *k*_max value and whether LTMF is used. All experiments were performed on a DecStation 5000/200, running a C code, which was generated using the optimizing (−O) option.

Table 2 shows the overwhelming superiority of the results obtained by using the long term memory function introduced in §3.2. The TS method with long term memory was able to optimally solve the example problem on every solution trial for each value of *k*_max, with the exception of a single trial (when tabu_size

Figure 2 Telecommunications Network for Example Problem

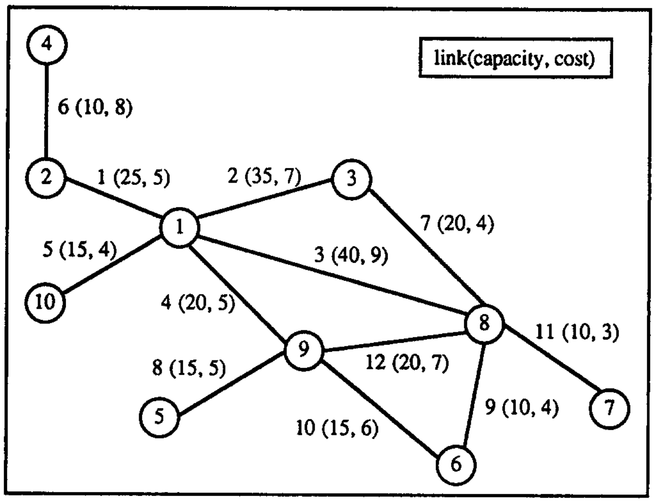

Table 1 Call Table for Example Problem

Call	From	To	Demand	Revenue	Call	From	To	Demand	Revenue
1	1	3	10	420	11	4	6	6	850
2	1	8	7	380	12	6	3	3	200
3	1	6	6	400	13	7	10	5	370
4	1	5	6	390	14	8	2	6	500
5	2	7	5	500	15	8	10	5	340
6	2	6	5	490	16	8	5	2	120
7	2	5	7	400	17	9	2	6	460
8	3	10	2	150	18	9	3	8	450
9	3	4	4	450	19	10	6	5	360
10	3	5	8	500	20	10	2	5	170

Table 2 Summary of Results for the Example Problem*

	tabu_size			
LTMF	S	M	L	D
OFF				
NOS	0	0	0	0
PDO	3 8	3.8	1.4	0.9
ITB	18	14	39	203
ON				
NOS	5	4	5	5
PDO	0.0	0.01	0 0	0.0
ITB	3186	2429	4167	4315

* Each cell represents outcomes for five trials using different *k*_max values.

= *M* and *k*_max = 6, resulting in a near optimal objective value of 4,988). By contrast, when the long term memory was switched OFF, the method did not succeed in finding an optimal solution on any of the trials, within the specified number of iterations. However, the average solution quality for the dynamic list size is superior to that obtained for any of the three fixed sizes under this option, confirming the general value of dynamic lists.

The average time to complete the search increases with the *k*_max value, as shown by Table 3. There is also a difference of about one second between the time required when LTMF is used and when this function is switched OFF. It is important to note that these times correspond to solution trials of 15,000 iterations, and that the best solutions are typically found within 5,000 iterations. Therefore, the number of iterations can be set much smaller without inhibiting the ability to obtain optimal solutions. In fact, for *k*_max = 2 and using LTMF, our TS method finds optimal solutions for every tabu size setting (i.e. S, M, L, and D), on an average of 325 iterations, corresponding to 0.13 CPU seconds.

Table 3 Average CPU Seconds for Solution Attempts With and Without LTMF

	*k*_max				
LTMF	2	3	4	5	6
OFF	5.4	6.6	7.8	8.8	9 6
ON	5.8	7.4	8.7	9.5	10.2

6. The BWP Problem Without Link Costs

An important variant of the BWP problem takes place when link costs are all set equal to zero. In this case, the objective is to maximize total revenue (i.e., the sum of the revenues of all calls in **A**). In practice link costs often can be neglected (especially for already existing telecommunication networks), and therefore this alternate formulation of the BWP problem is considered by practitioners to be more realistic for some types of applications (Qiu 1991). Our method can also be used to find high quality solutions to instances of the BWP problem without link costs; however, two minor modifications are appropriate. The changes occur in the starting solution procedure and the selection of a best move during the search.

6.1. A Modified Starting Solution Generator

The starting solution procedure for the problem with link costs selects, from the set of feasible paths available at the current state solution state, the most profitable path to assign the call under consideration. When no costs are associated with the links in the network, the concept of profitability is lost, i.e., a feasible path is as good as any other one in terms of revenue. To be able to differentiate among paths, the concept of *slack* is introduced. Let call *i* be the current call under consideration. Then for each feasible path *h* a slack value is calculated as follows:

$$\text{slack}(i, h) = \sum_{j \in L_h} (b_j - f_j - r_i).$$

Note that a feasible path requires that $b_j - f_j - r_i \geq 0$ for all $j \in L_h$. The starting procedure then selects the feasible path with the smallest slack value. This choice rule forces the selection of paths that are either smaller in length (measured in number of links) or for which the bandwidth requirements of the call are closer to exhausting the capacities of the corresponding links. Furthermore, there is a particular class of problems that this constructive heuristic will optimally solve (e.g., problem 2 on Table 4).

6.2. A Choice Rule for the Best Move

The best move selection in our original method is based on a move value calculation that considers the cost difference of changing the path assignment for a particular call (see §3.2). In the new model, move values that make reference to the objective function provide only limited information in the local search of a given neighborhood. Consider the move of call i from path g to a feasible path h. Then the value of this move, move_value(i, h), is equal to

$$\begin{array}{rl} v_i & \text{if } g = 0 \text{ and } 0 < h \le k_i, \\ -v_i & \text{if } 0 < g \le k_i \text{ and } h = 0, \\ 0 & \text{if } 0 < g \le k_i \text{ and } 0 < h \le k_i. \end{array}$$

Therefore, evaluating a neighborhood in this way will result in a large percentage of moves having a zero value. Once more we make use of the *slack* concept to modify our previous definition of a penalized move value (pmv). In this case, we are interested in rewarding moves that increase the total available capacity while keeping the same (or almost the same) total revenue. The penalized move value is then given by the following expression:

$$\text{pmv}(i, h) = \text{move\_value}(i, h) + r_i * (|L_g| - |L_h|) - \alpha * \text{freq}(i, h)$$

where $|X|$ is the cardinality of the set X, and $\alpha = 10$ if move_value$(i, h) \le 0$, otherwise $\alpha = 0$. The best move is the candidate move with the largest penalized move value. As before, this rule is only used when improving nontabu moves are not available, otherwise the most improving move is selected.

7. Final Computational Experiments

Our final experimentation consists of comparing the solutions obtained by our methods with other existing optimal and heuristic procedures. The instances used in our experiments were generated to simulate demands in telecommunication networks that resemble those found in practice. (The problem generator belongs to U S WEST Advanced Technologies and the problem instances were obtained by courtesy of Yuping Qiu (Qiu 1991).) The first set consists of ten instances of the BWP problem without link costs. Table 4 shows the problem numbers, their size (number of nodes and calls), the best solution found by running LINDO for extended time periods (several days for some of the larger problems), the best solution obtained by different search methods developed in U S WEST (Qiu 1991), the solutions found by a *permutation based* TS method developed in (Anderson et al. 1990), and the results obtained by our TS procedure. (A permutation-based approach to the BWP problem considers a trial solution to be an ordering of the calls. An ordering is mapped into an assignment by routing each call through the best available path subject to the resources remaining in the network after all previous calls have been routed.)

The results reported in Table 4 for our method were obtained by setting k_max to a value of $n/2$ for all solution trials (except for problem 2) and by using dynamic list sizes in combination with the long term memory function. All instances were run a maximum of 10,000 iterations resulting in an average CPU time of 19.9 seconds. The best solutions were found on an average of 1,909 iterations, corresponding to 4.4 CPU seconds. Problem 2 is optimally solved in 0.11 CPU seconds by our starting solution procedure, when k_max is set to a value of n. This problem consists of 20 calls from node 1 to 2 with capacity requirements ranging from 10 to 200 (in increments of 10), and the network is structured so that there are only 20 unique paths between these two nodes with the same capacities as the calls. The revenue for the call with capacity requirements of 10 is 200, for the one with demand of 20 is 190, and so on. This type of problems is especially hard for search procedures that are only directed by the revenue differences, since calls with large revenue (and

Table 4 **Results of Experiments for Networks Without Link Costs**

Number	Size	LINDO*	Best U S WEST Heuristic	Permutation Based TS	Glover & Laguna
1	(10, 20)	7,540	7,540	7,540	7,540
2	(21, 20)	2,100	2,100	2,100	2,100
3	(31, 50)	13,280	13,380	13,270	13,550
4	(10, 20)	2,955	2,955	2,885	2,955
5	(16, 20)	2,315	2,345	2,345	2,345
6	(17, 30)	8,850	9,010	9,010	9,010
7	(20, 40)	11,000	11,160	11,160	11,000
8	(12, 36)	12,810	12,810	12,810	12,810
9	(12, 24)	5,780	5,780	5,780	5,780
10	(14, 28)	940	970	970	970

* The IP formulation consisted of a maximum of six paths per call.

small capacity requirements) are more likely to be incorrectly assigned to large capacity paths.

The TS method that we adapted for the solution to the BWP problem without link costs compares well with highly specialized procedures specifically designed for this class of problems. In all but one instance (problem 7) our results are at least as good as the ones obtained by the specialized permutation-based TS method. Our procedure is also able to provide the best known solution to problem 3, improving upon the best lower bound found by LINDO and the U S WEST heuristics. In terms of computational effort our method is also attractive, considering that the permutation-based approach requires on the average 16.5 CPU seconds to find the best solutions on a VAX 8800. It is also important to note that the computer times required by our approach were more than two orders of magnitude less than those required by LINDO (CPU seconds versus many hours or even days).

An additional experiment with networks without link costs was performed using a set of eight problems introduced in (Anderson et al. 1990). These instances have numbers of nodes ranging from 10 to 30 and numbers of calls ranging from 9 to 46. Our TS procedure matched all the best known solutions to these problems in less than 4 CPU seconds, and more importantly, this outcome was achieved using the same parameter settings as before, with no fine tuning.

The ten problems in the second set contain networks with link costs, which cannot be handled by the specialized permutation based procedure. The experiments with these problems were performed using the same settings as before, i.e., the *k*_max value was set to $n/2$, dynamic tabu lists were used, and the long term memory function was activated. Table 5 compares the best solution known to these problems with the solution obtained by our TS method. This table also shows the CPU seconds required for each solution attempt consisting of 10,000 iterations. In all but one case (problem 7) our TS procedure succeeds in improving (or matching) the previous best known solution to each problem instance.

The best known solutions presented in Table 5 were found at the expense of computer time. Specifically, the small problems ($q \leq 30$) required between 3 to 4 CPU hours on a DEC 3100, and the large problems were run for several days (Qiu 1991). For this last experiment, we modified our move evaluation function to handle the fixed cost associated with each link in problem 2. This modification did not seem to affect the quality of the solution obtained by our TS approach.

Table 5 **Results of Experiments for Networks with Link Costs**

			Glover & Laguna	
Number	Size	Best Known	Solution	CPU Seconds
1	(10, 20)	6,076	6,076	7.70
2*	(21, 20)	1,910	1,921	32.56
3	(31, 50)	6,326	6,393	43.07
4	(10, 20)	2,779	2,784	7.59
5	(16, 20)	2,288	2,288	11.41
6	(17, 30)	7,990	8,039	19.03
7	(20, 40)	9,550	9,539	33.60
8	(12, 36)	11,220	11,349	15.89
9	(12, 24)	4,554	4,555	10.75
10	(14, 28)	638	656	6.01

* This problem contains fixed costs that are different for each link.

8. Conclusions

We have identified the design of a TS procedure for the bandwidth packing problem, and have determined the importance of a long term memory function for obtaining the highest quality solutions under various parameter settings. By introducing reference to a slack concept, our procedure is successful for problems both with and without link costs. The method is also highly robust; single parameter settings based on preliminary experimentation with a single problem succeeded in obtaining the solutions reported to all other problems, without further tuning or modification. Computation times required by our method are on the average somewhat less than those required by the specialized permutation based approach, and they appear to be at least two orders of magnitude better than those of the LINDO integer programming package, and these differences increase as larger problems are examined. Certainly more study is warranted, as by identifying more sophisticated (adaptive) ways to generate the parameter α for exploiting frequency based memory. We anticipate the concepts introduced here will also be useful in adaptation to more complex classes of problems, where a pool of best candidate paths cannot be generated appropriately in advance, but must be redetermined according to the current solution state.[1]

[1] This research was supported in part by the Joint Air Force Office of Scientific Research and Office of Naval Research Contract No. F49620-90-C-0033 at the University of Colorado.

References

Anderson, A., K. F. Jones and J. Ryan, "Path Assignment for Call Routing: An Application of Tabu Search," *Annals of Oper. Res.*, 41 (forthcoming).

Cox, L. A., L. Davis and Y. Qiu, " 'Intelligent' Probabilistic Search for Routing Strategies in Circuit-Switched Telecommunications Networks," Research Report, U S West Advanced Technologies, 1990.

Glover, F., "Tabu Search—Part I," *ORSA J. Computing*, 1 (1989), 190–206.

———, "Tabu Search for Call Routing—Part I," Research Report, Graduate School of Business, University of Colorado at Boulder, Boulder, CO, 1990.

Hertz, A. and D. de Werra, "The Tabu Search Metaheuristic: How We Used It," *Ann. Math. and Artificial Intelligence*, 1 (1990), 111–121.

Lawler, E. L., *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

Laguna, M. and F. Glover, "Integrating Target Analysis and Tabu Search for Improved Scheduling Systems," to appear in *Expert Systems with Applications: An International Journal.*

Qiu, Y., U S WEST Advanced Technologies, private communication, 1991.

Skorin-Kapov, J., "Extensions of a Tabu Search Adaptation to the Quadratic Assignment Problem," Harriman School Working Paper 90-006, State University of New York at Stony Brook, Stony Brook, NY, 1991.

Taillard, E., "Robust Tabu Search for the Quadratic Assignment Problem," *Parallel Computing*, 17 (1990), 443–455.

Accepted by Thomas M. Liebling; received June 1991. *This paper has been with the authors* 3 *weeks for* 1 *revision.*