FACULTAD DE INGENIERÍA MECÁNICA Y ELÉCTRICA

UNIVERSIDAD AUTÓNOMA DE NUEVO LEÓN

Facultad de Ingeniería Mecánica y Eléctrica

Universidad Autónoma de Nuevo León

Computational Experience with Heuristics for the Multiple Knapsack Problem.

Team E Final Project

2109524	Fátima Sofía Aranda Cruz
1962995	Carlos Abraham Gallardo Treviño
2078017	Andrés Isaac Franco Benavides
1894607	Gerardo Gael Peña Cital

1. Introduction.

Knapsack problems represent a class of combinatorial optimization issues that arise in various real-world scenarios. These problems involve selecting a subset of items, each with a given weight and value, to maximize the total value without exceeding a specified weight limit. The complexity of knapsack problems stems from the need to balance multiple constraints, such as weight, value, and sometimes additional factors like the number of items in certain categories or personal preferences. This variability necessitates diverse solution methods tailored to specific contexts and requirements.

In this paper, we focus on the Multiknapsack problem, a variant where multiple knapsacks are used, each with its own capacity constraint. Solving the Multiknapsack problem requires strategies that consider the allocation of items across multiple knapsacks to maximize the total value. To address this, we employ a heuristic approach based on the value-to-weight ratio, inspired by the methodology of Martello and Toth. This heuristic prioritizes items with the highest value-to-weight ratio, providing an initial solution that leverages the most valuable items efficiently.

To enhance the initial solution, we incorporate the 2-opt local search algorithm. This method iteratively swaps pairs of items between knapsacks, seeking to improve the overall solution. Additionally, we utilize First Found algorithms, which expedite the local search process by accepting the first improvement encountered. This combination of heuristic and local search techniques allows us to achieve a more refined and effective solution to the Multiknapsack problem.

Our approach begins with a thorough examination of the basic structure of knapsack problems, including various examples and procedures. We then delve into the specifics of the Martello and Toth heuristic, analyzing its advantages, nuances, and potential drawbacks. By understanding these aspects, we aim to demonstrate the practical application of heuristics in solving complex optimization problems in a meaningful and impactful way.

2. Problem description.

For the Multiple Knapsack Problem, the following variables are known and needed in order to get a feasible solution (Marcelo & Toth, 1990):

- Quantity of items (n).
- Quantity of knapsacks ($m \leq n$)
- Profit of item j (p_j).
- Weight of item (W_j).
- Capacity of Knapsack i (C_i).

Based on these variables, decisions will be made regarding which items to add to certain knapsack, considering that each knapsack has a different capacity, and each item has a different value, in order to find the assignment of items to knapsacks that maximizes the total value.

However, this process is limited by the following constraints:

- Each item can be allocated to at most one knapsack.
- The knapsack capacity cannot be exceeded.

Likewise, this will be achieved by following the next objective function (Lalami et al., 2012):

$$\text{Maximise: } \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij},$$

in which N represents the number of items, M the number of knapsacks, p_j is the value of item j and X_{ij} is a binary decision variable that indicates whether the item j is selected.

3. Problem example.

An example in real life that can be modeled using the Multiple Knapsack Problem is the distribution of goods by a logistics company to various delivery trucks.

Here's how the variables and constraints could apply:

- Quantity of items: Represents the different types of goods that need to be distributed.
- Quantity of knapsacks (delivery trucks): Represents the number of available trucks for distribution.
- Profit of item j (pj): Represents the value or importance of each type of goods.
- Weight of item (Wj): Represents the physical weight or volume of each type of goods.
- Capacity of Knapsack i (Ci): Represents the maximum weight or volume each truck can carry.

The objective would be to maximize the total value of goods distributed while ensuring that each truck doesn't exceed its capacity and that each type of goods is allocated to only one truck. In addition, the decision variables (Xij) would indicate whether a specific type of goods is loaded onto a particular truck. By optimizing the assignment of goods to trucks while considering their values and the capacity constraints of each truck, the logistics company can efficiently distribute goods to maximize profit and ensure timely deliveries.

The above was a specific way of projecting this type of problem in a real-life situation, for which there are a variety of solutions that can be applied. Below are listed the different ways to solve it, each with an accompanying explanation of how this would be represented algorithmically in a high-level language, for which we have chosen to use Python due to its dedication to data management. Furthermore, in this case, for the intention of illustrating, a small system has been implemented in all the methods:

Problem: Suppose we have a knapsack with a capacity of 50 units and a set of items each with a specific weight and value.

Items:

Item 1: Weight = 10, Value = 60

Item 2: Weight = 20, Value = 100

Item 3: Weight = 30, Value = 120

Dynamic Programming

A method of solving this issue which uses certain cases to generate a result using a set of operations, which in turn have certain solutions to these cases. This is the type of method which are used on local search to attain better results to before made heuristics and is with it a method which improves on what was done before hand.

Steps:

1. Create a 2D array ‘dp’ where ‘dp[i][w]’ represents the maximum value that can be attained with the first ‘i’ items and a weight limit ‘w’.
2. Initialize the array with zeros.
3. Fill the array using the relation:

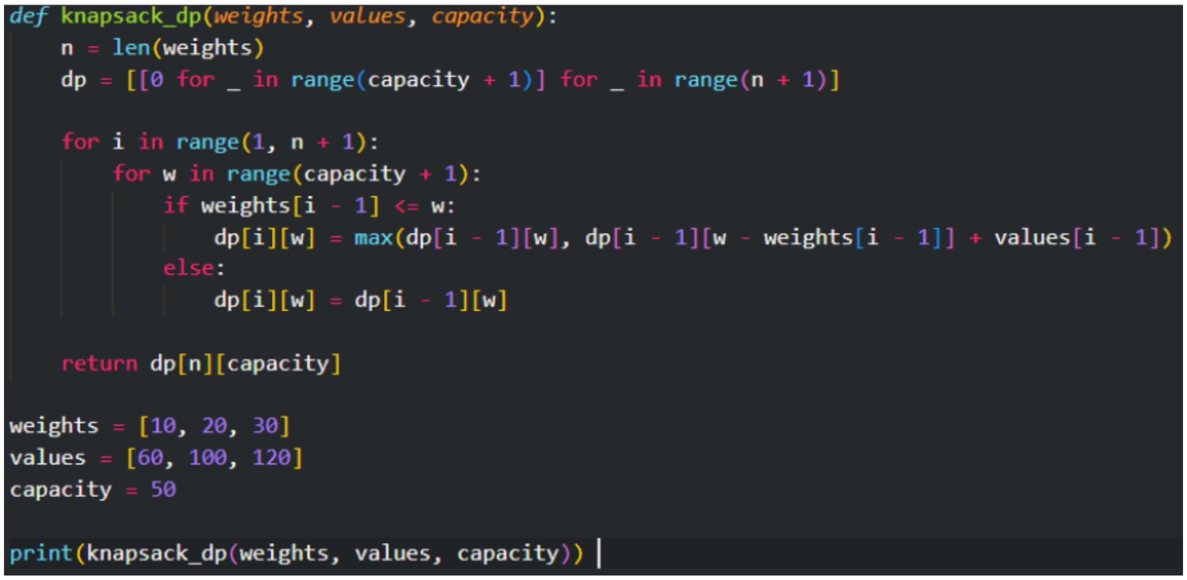
$$dp[i][w] = \max(dp[i-1][w], dp[i-1][w - weight_i] + value_i)$$

$$if\ weight_i \leq w$$

Solution:

• Initialize ‘dp’ with dimensions (n+1) × (W+1) where n is the number of items and W is the knapsack capacity.

• Fill the ‘dp’ table using the recurrence relation. The maximum value that can be achieved with a capacity of 50 is 220.



```python
def knapsack_dp(weights, values, capacity):
    n = len(weights)
    dp = [[0 for _ in range(capacity + 1)] for _ in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(capacity + 1):
            if weights[i - 1] <= w:
                dp[i][w] = max(dp[i - 1][w], dp[i - 1][w - weights[i - 1]] + values[i - 1])
            else:
                dp[i][w] = dp[i - 1][w]

    return dp[n][capacity]

weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50

print(knapsack_dp(weights, values, capacity))
```

Image 1. Example of dynamic programming using Python.

Using this method the maximum value that can be achieved with a capacity of 50 is 220.

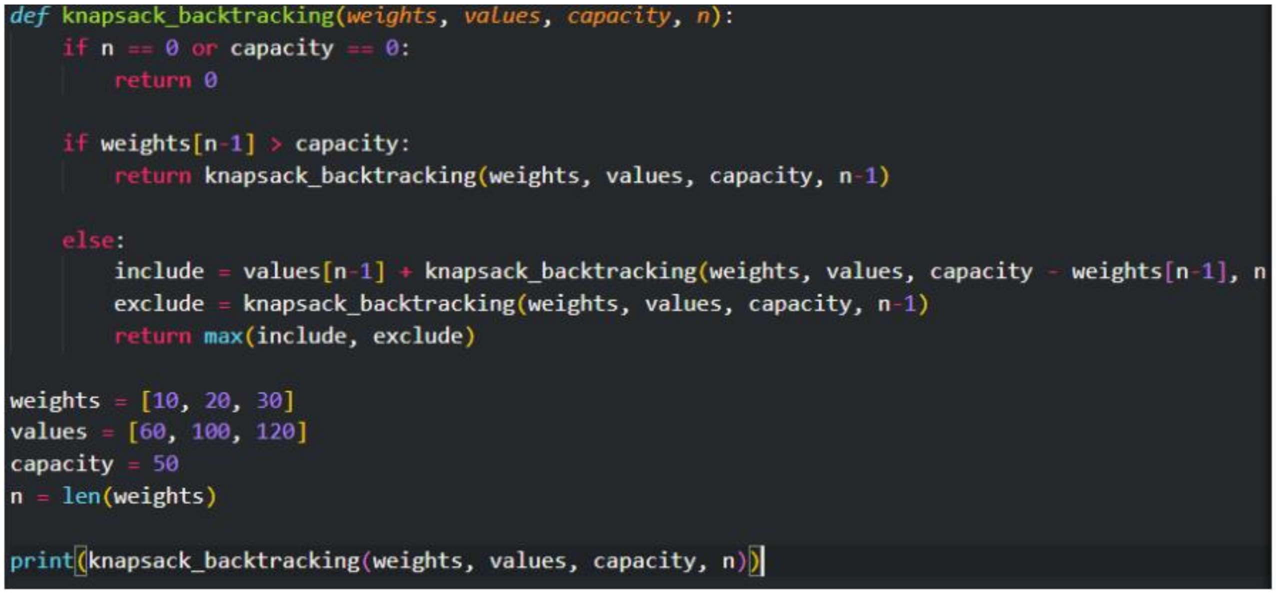
Backtracking Method

This method uses backtracking to see new paths towards a result by changing cases when a less optimal path is taken with small modifications on a predefined previous iteration to get a better result by seeing all paths available, this method with other brute force methods are unreliable for quick generation of solutions which our problem needs but are an option which has some need of analyzing.

Steps:

1. Start with an empty knapsack.
2. Recursively explore both possibilities for each item: including it or excluding it.
3. Backtrack if the current total weight exceeds the capacity.

Solution:



```
def knapsack_backtracking(weights, values, capacity, n):
    if n == 0 or capacity == 0:
        return 0

    if weights[n-1] > capacity:
        return knapsack_backtracking(weights, values, capacity, n-1)

    else:
        include = values[n-1] + knapsack_backtracking(weights, values, capacity - weights[n-1], n
        exclude = knapsack_backtracking(weights, values, capacity, n-1)
        return max(include, exclude)

weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50
n = len(weights)

print(knapsack_backtracking(weights, values, capacity, n))
```

Image 2. Example of backtracking method using Python.

The backtracking approach also yields a maximum value of 220 for the given capacity.

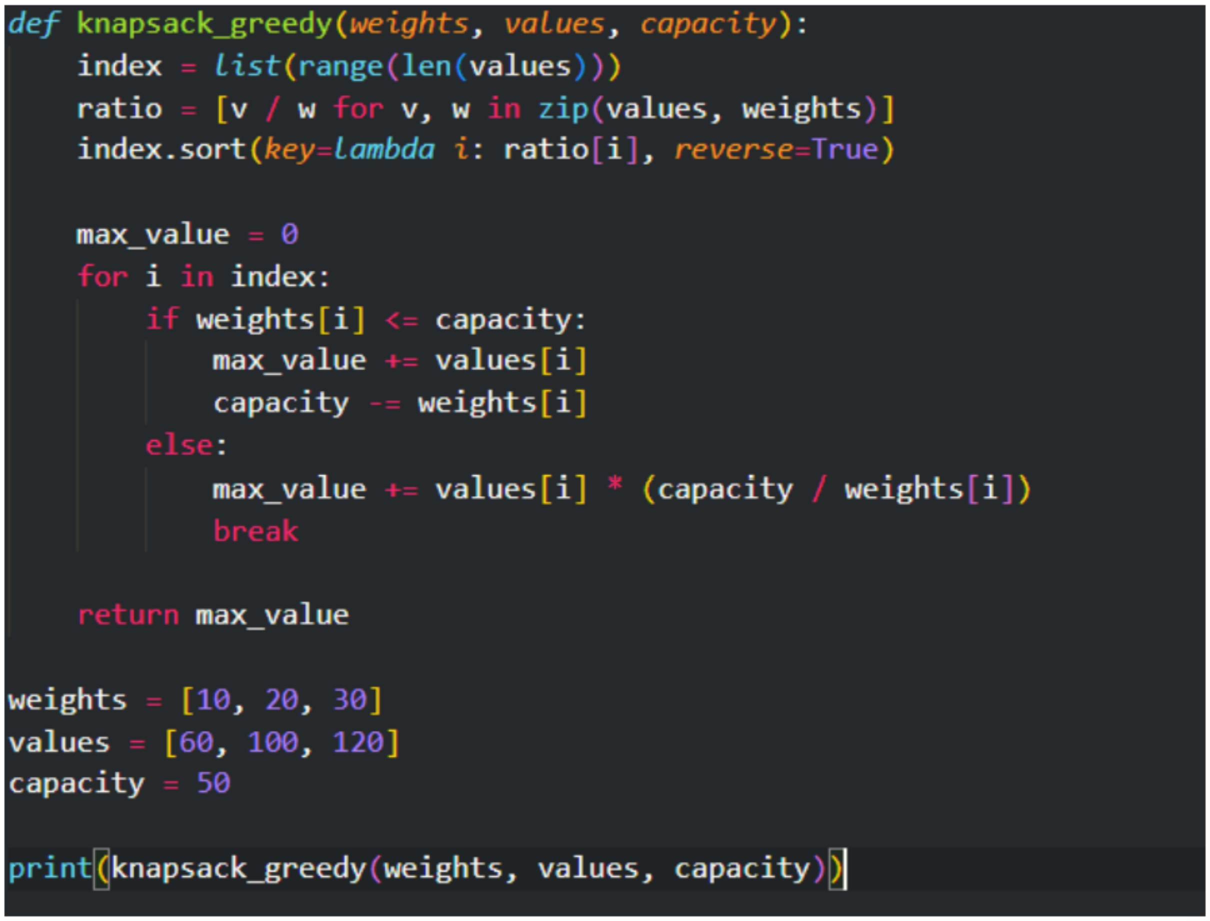
Greedy Function

Method in which a desired trait is put at the upmost importance so with it this method can be used to generate the result by doing a series pf modification to find better results by increasing this number, on this is the method which uses nonlinear search heuristics and is the first use cases for the problem which we are focusing on.

Steps:

1. Calculate the value-to-weight ratio for each item.
2. Sort items based on this ratio in descending order.
3. Add items to the knapsack starting from the highest ratio until the capacity is reached.

Solution:



```python
def knapsack_greedy(weights, values, capacity):
    index = list(range(len(values)))
    ratio = [v / w for v, w in zip(values, weights)]
    index.sort(key=lambda i: ratio[i], reverse=True)

    max_value = 0
    for i in index:
        if weights[i] <= capacity:
            max_value += values[i]
            capacity -= weights[i]
        else:
            max_value += values[i] * (capacity / weights[i])
            break

    return max_value

weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50

print(knapsack_greedy(weights, values, capacity))
```

Image 3. Example of a greedy function using Python.

The greedy approach results in a value of 240, which is higher than the optimal value for this specific problem, indicating the greedy approach might not always yield the optimal solution but provides a quick approximation.

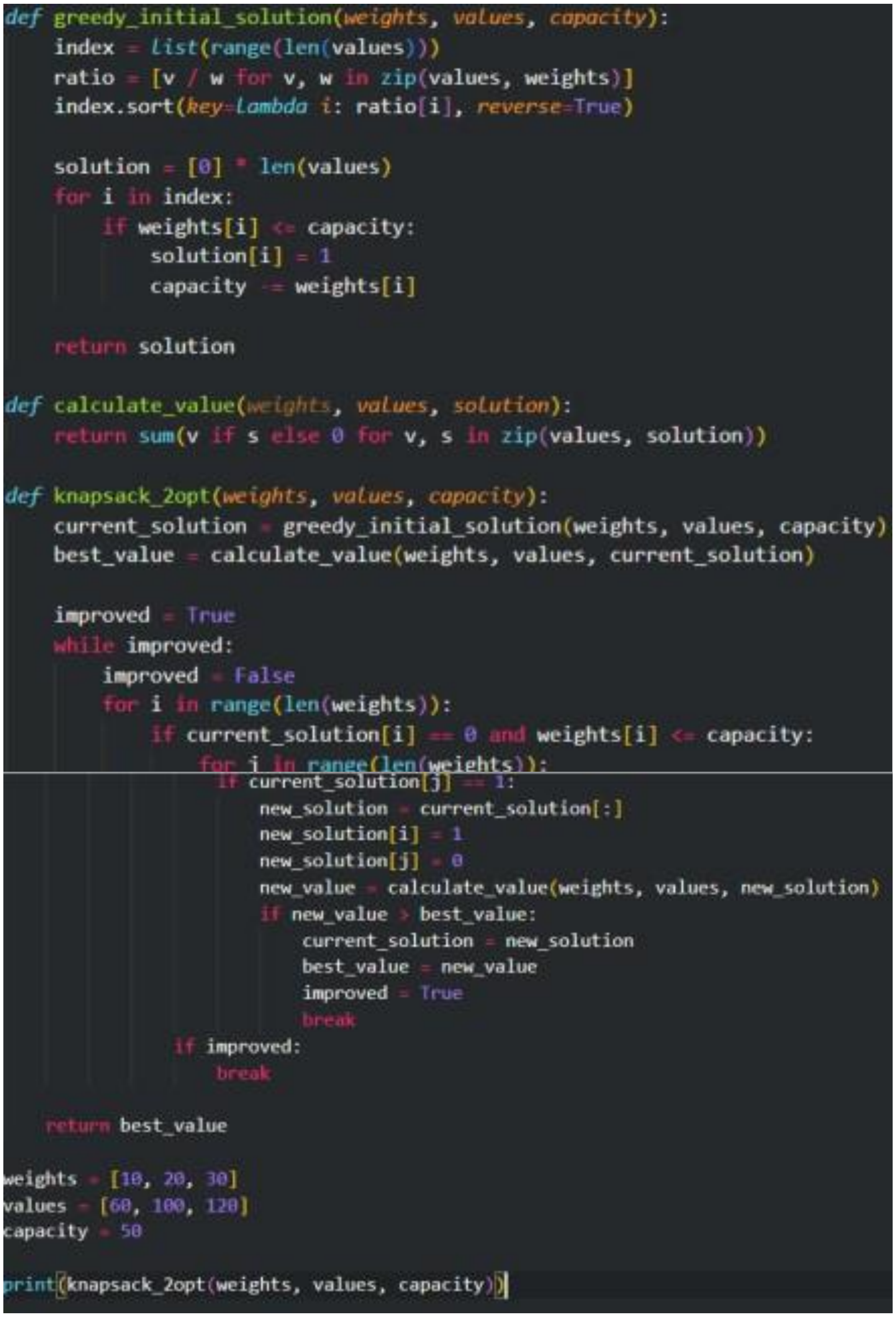
Mixed Approach Using Greedy and 2-OPT

Steps:

1. Use a greedy algorithm to get an initial feasible solution.

2. Apply the 2-OPT technique to iteratively improve the solution.

Solution:



```python
def greedy_initial_solution(weights, values, capacity):
    index = list(range(len(values)))
    ratio = [v / w for v, w in zip(values, weights)]
    index.sort(key=lambda i: ratio[i], reverse=True)

    solution = [0] * len(values)
    for i in index:
        if weights[i] <= capacity:
            solution[i] = 1
            capacity -= weights[i]

    return solution

def calculate_value(weights, values, solution):
    return sum(v if s else 0 for v, s in zip(values, solution))

def knapsack_2opt(weights, values, capacity):
    current_solution = greedy_initial_solution(weights, values, capacity)
    best_value = calculate_value(weights, values, current_solution)

    improved = True
    while improved:
        improved = False
        for i in range(len(weights)):
            if current_solution[i] == 0 and weights[i] <= capacity:
                for j in range(len(weights)):
                    if current_solution[j] == 1:
                        new_solution = current_solution[:]
                        new_solution[i] = 1
                        new_solution[j] = 0
                        new_value = calculate_value(weights, values, new_solution)
                        if new_value > best_value:
                            current_solution = new_solution
                            best_value = new_value
                            improved = True
                            break
                if improved:
                    break

    return best_value

weights = [10, 20, 30]
values = [60, 100, 120]
capacity = 50

print(knapsack_2opt(weights, values, capacity))
```

Image 3. Example of a greedy function using Python.

This mixed approach first finds a feasible solution using the greedy method and then iteratively improves it using a local search method (2-OPT). This approach also yields a maximum value of 220, demonstrating an efficient combination of methodologies.

4. Description of heuristics.

In addressing the Multiknapsack problem, we employed a heuristic approach that involves ordering the items from highest to lowest value. This heuristic is based on the premise that prioritizing items with the highest value maximizes the total value in the knapsacks. By sorting the items this way, we can initially allocate the most valuable items to the knapsacks, thereby providing a strong starting solution.

To further refine this initial solution, we utilized the 2-opt local search algorithm. The 2-opt method works by iteratively swapping two items between the knapsacks and assessing whether this exchange improves the overall solution. This process continues until no further improvements can be made, ensuring that we reach a locally optimal solution. However, in order to minimize computational resources and time, we implemented First Found algorithms, which expedite the local search process by accepting the first improvement found during the search, thus speeding up convergence to an improved solution.

By combining the value-based heuristic with the 2-opt local search and First Found algorithms, we achieve a more effective solution to the Multiknapsack problem. The heuristic provides a solid foundation by leveraging high-value items, while the 2-opt local search and First Found algorithms fine-tune the allocation to optimize the total value across the knapsacks. This approach not only enhances the solution's quality but also demonstrates the practical utility of integrating heuristic methods with local search techniques in solving complex combinatorial optimization problems.

5. Computational work.

To evaluate the performance of the algorithms and heuristics, we developed three programs using the C programming language. The first program was designed to create test instances based on specified parameters, including the number of knapsacks, the number of items, and ranges for item values and weights. This program generated instances by assigning random weights and values to each item within the given ranges.

The second program implemented the heuristic approach. This program processed the data generated by the first program, which was stored in files with a .dat extension. It applied the value-to-weight ratio heuristic to order the items and allocate them to the knapsacks, forming an initial solution.

The third program focused on the local search algorithm. Using the initial solutions generated by the heuristic program, this program performed iterative improvements by swapping pairs of items between knapsacks to enhance the overall solution. Like the heuristic program, it also read data from .dat files generated by the first program.

The testing process consisted of three segments, each involving 20 instances of varying sizes: small, medium, and large. The small instances included 5 knapsacks and 1,000 items, the medium instances had 500 knapsacks and 100,000 items, and the large instances comprised 5,000 knapsacks and 1,000,000 items. Each segment was designed to test the scalability and efficiency of the algorithms and heuristics under different conditions, ensuring a comprehensive evaluation of their performance across a range of scenarios. Next, the results will be showed according to their size.

Small instances

Sample	Final Combined Value	Execution's time
1	197467	0.00001
2	197613	0.000009
3	199786	0.00001
4	197897	0.000009
5	198882	0.000009
6	197538	0.000009
7	198818	0.00001
8	201641	0.000009
9	199846	0.00001
10	202666	0.000009
11	201382	0.00001
12	194691	0.00001
13	201699	0.00001
14	198879	0.000009
15	203363	0.000009
16	198727	0.000008
17	199369	0.00001
18	199473	0.00001
19	201963	0.00001
20	198943	0.00001

Table 1. Solutions after first heuristic (small instances).

Sample	Final Combined Value	Execution's time	Absolute improvement	Relative improvement
1	200220	0.86346	2753	1.39416%
2	200000	0.97508	2387	1.20792%
3	201678	0.93531	1892	0.94701%
4	201680	1.00281	3783	1.91160%
5	201486	1.01391	2604	1.30932%

6	199187	0.80393	1649	0.83478%
7	199826	0.75668	1008	0.50700%
8	204163	0.99691	2522	1.25074%
9	203729	1.04725	3883	1.94300%
10	204257	0.74694	1591	0.78504%
11	203277	0.94280	1895	0.94100%
12	196193	0.96368	1502	0.77148%
13	202757	0.81492	1058	0.52454%
14	200152	0.78398	1273	0.64009%
15	204765	0.74971	1402	0.68941%
16	202670	0.88750	3943	1.98413%
17	202120	0.77429	2751	1.37985%
18	202853	0.95343	3380	1.69446%
19	203528	0.88025	1565	0.77489%
20	201455	0.82404	2512	1.26267%
		Average	2267.65	1.13765%

Table 2. Solutions and rates of improvement after local search heuristic (small instances).

Corresponding to the results of the first group of instances, an average relative improvement of 1.14% was obtained from the application of local search, with a maximum improvement of 1.94%, while the smallest improvement was 0.51%. This is shown in the following graph, which displays the relative improvement for each instance in this group.

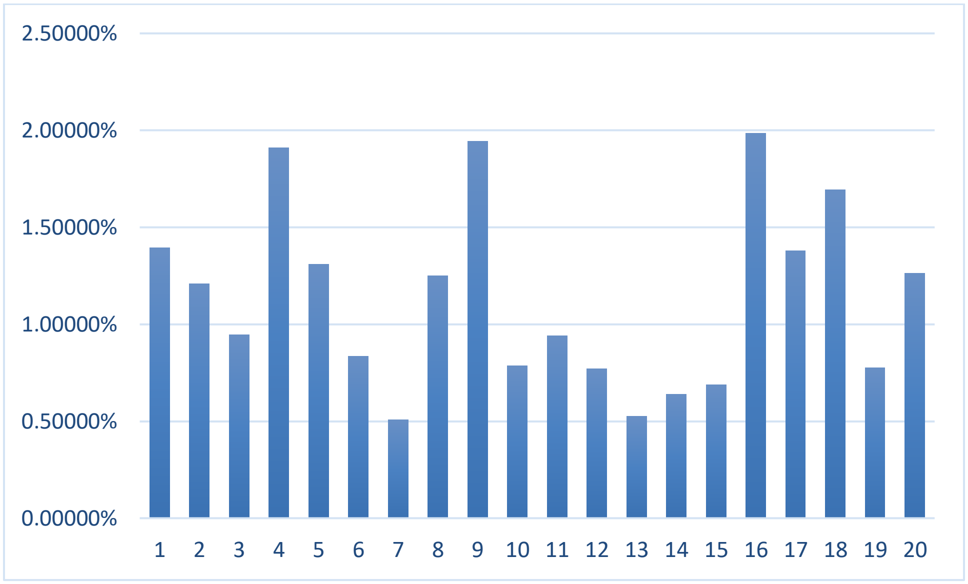


Figure 1. Relative improvement on each solution after local search (small instances).

This can be further supported by the following graph, which shows the improvement more clearly. In this graph, each line represents the set of final solutions, with the blue line representing the set before applying local search and the red line corresponding to the set after this process. In this way, the increase in the final value of each solution is evident in each segment of the graph.

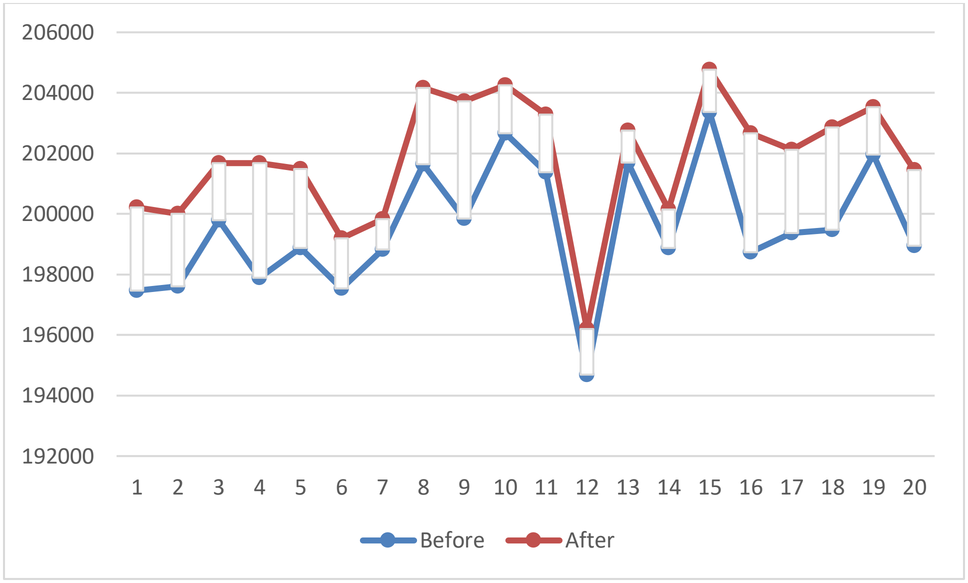


Figure 2. Final total value before and after the local search (small instances).

Medium instances

Sample	Final Combined Value	Execution's time
1	16831771	0.000009
2	16872269	0.00001
3	16848033	0.000009
4	16829585	0.00001
5	16827491	0.000009
6	16809195	0.000009
7	16826197	0.00001
8	16837344	0.000009
9	16789786	0.00001
10	16837587	0.000009
11	16839615	0.00001
12	16831825	0.00001
13	16860155	0.000009
14	16854953	0.00001
15	16855415	0.00001
16	16776632	0.00001
17	16818979	0.00001
18	16829969	0.000009

19	16878397	0.00001
20	16836469	0.000009

Table 3. Solutions after first heuristic (medium instances).

Sample	Final Combined Value	Execution's time	Absolute improvement	Relative improvement
1	17065027	37.564	233256	1.38581%
2	17103828	33.898	231559	1.37242%
3	16946329	37.493	98296	0.58343%
4	17024333	35.014	194748	1.15718%
5	17033354	34.443	205863	1.22337%
6	16967279	36.900	158084	0.94046%
7	17002226	39.100	176029	1.04616%
8	17021088	35.714	183744	1.09129%
9	16902372	38.285	112586	0.67056%
10	17020173	36.055	182586	1.08440%
11	17058149	36.919	218534	1.29774%
12	17033446	34.944	201621	1.19786%
13	17076211	35.492	216056	1.28146%
14	17051638	34.484	196685	1.16693%
15	16958923	34.732	103508	0.61409%
16	16905263	39.135	128631	0.76673%
17	17027522	39.056	208543	1.23993%
18	17043154	35.411	213185	1.26670%
19	17015559	37.381	137162	0.81265%
20	16969956	36.709	133487	0.79284%
		Average	176708.15	1.04960%

Table 4. Solutions and rates of improvement after local search heuristic (medium instances).

Based on the outcomes of the initial set of instances, an average relative enhancement of 1.05% was attained following the implementation of local search. The improvement ranged from a minimum of 0.58% to a maximum of 1.39%. These variations are graphically depicted below, showcasing the relative enhancement achieved for each instance within this dataset.

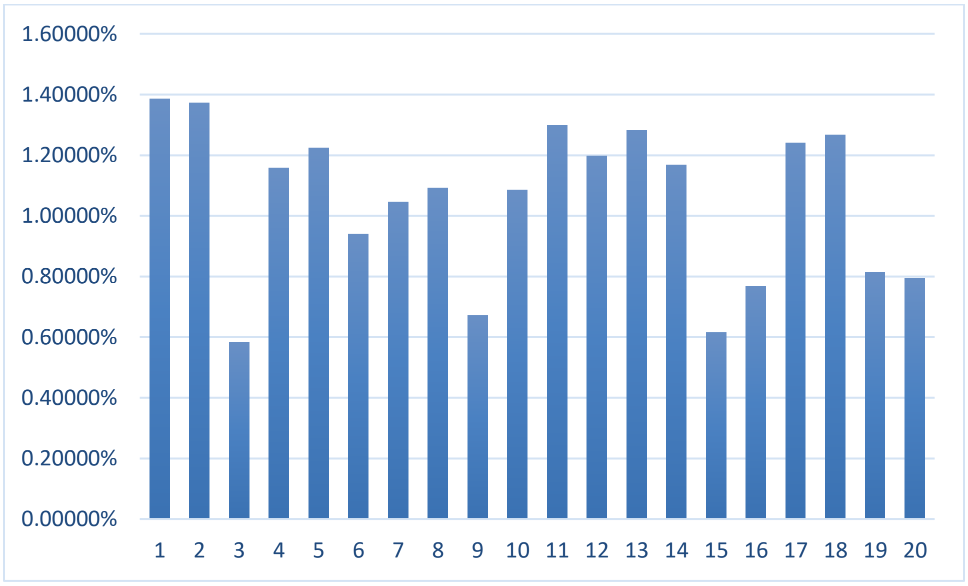


Figure 3. Relative improvement on each solution after local search (medium instances).

Further validation is provided through the subsequent visualization, which offers a clearer representation of the enhancement. Each line within this graph delineates the final solution set, following the same structure as the graphic from the group of small instances. Consequently, the augmented value of each solution becomes apparent across all segments of the graph.

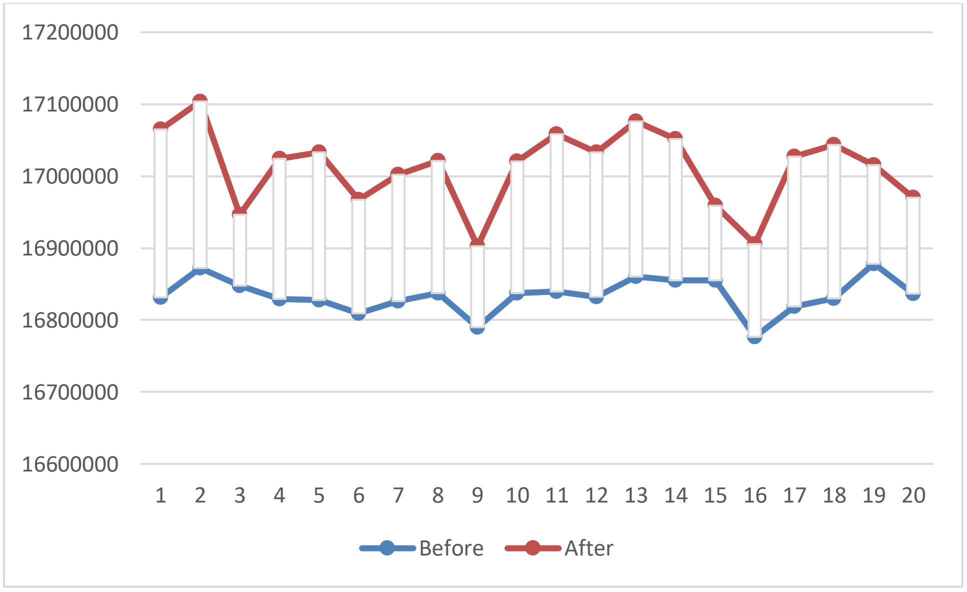


Figure 4. Final total value before and after the local search (medium instances).

Large instances

Sample	Final Combined Value	Execution's time
1	149997809	0.03964
2	149994524	0.03964
3	149914887	0.03964
4	149921792	0.03964
5	149949880	0.03963
6	149921916	0.03964
7	149932848	0.03965
8	149945940	0.03964
9	149989870	0.03964
10	149999965	0.03965
11	149983950	0.03964
12	149934878	0.03964
13	149974851	0.03964
14	149912941	0.03965
15	149943835	0.03964
16	149948866	0.03964
17	149929943	0.03964
18	149932919	0.03964
19	149923952	0.03963
20	149934953	0.03964

Table 5. Solutions after first heuristic (large instances).

Sample	Final Combined Value	Execution's time*	Absolute improvement	Relative improvement
1	150145477	600.00	147668	0.09845%
2	150118676	600.00	124152	0.08277%
3	150154933	600.00	240046	0.16012%
4	149994762	600.00	72970	0.04867%
5	150196038	600.00	246158	0.16416%
6	150033357	600.00	111441	0.07433%
7	150017212	600.00	84364	0.05627%
8	150178623	600.00	232683	0.15518%
9	150147812	600.00	157942	0.10530%
10	150266410	600.00	266445	0.17763%
11	150156796	600.00	172846	0.11524%
12	150163486	600.00	228608	0.15247%
13	150060339	600.00	85488	0.05700%
14	149995484	600.00	82543	0.05506%

15	150237468	600.00	293633	0.19583%
16	150230176	600.00	281310	0.18760%
17	150014843	600.00	84900	0.05663%
18	150209072	600.00	276153	0.18418%
19	150212592	600.00	288640	0.19252%
20	150158662	600.00	223709	0.14920%
		Average	185084.95	0.12343%

Table 6. Solutions and rates of improvement after local search heuristic (large instances).

Based on the results obtained from the initial set of instances, an average relative progress of 0.05% was recorded after applying the local search technique. The range of progress varied from a minimum of 0.58% to a maximum of 0.2%. These variations are visually depicted below, illustrating the relative progress achieved for each instance within this dataset.

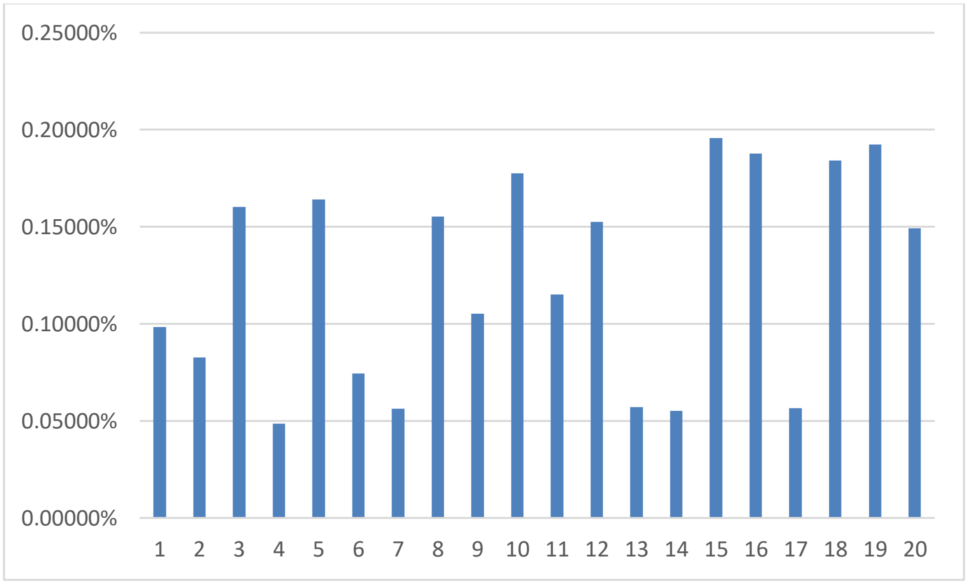


Figure 5. Relative improvement on each solution after local search (large instances).

Further confirmation is provided through the subsequent graphical representation, which offers a clearer visualization of the progress. Each line within this graph represents the final solution set, with the initial solution set depicted in blue and the modified solution set shown in red. Consequently, the increased value of each solution is evident across all segments of the graph.

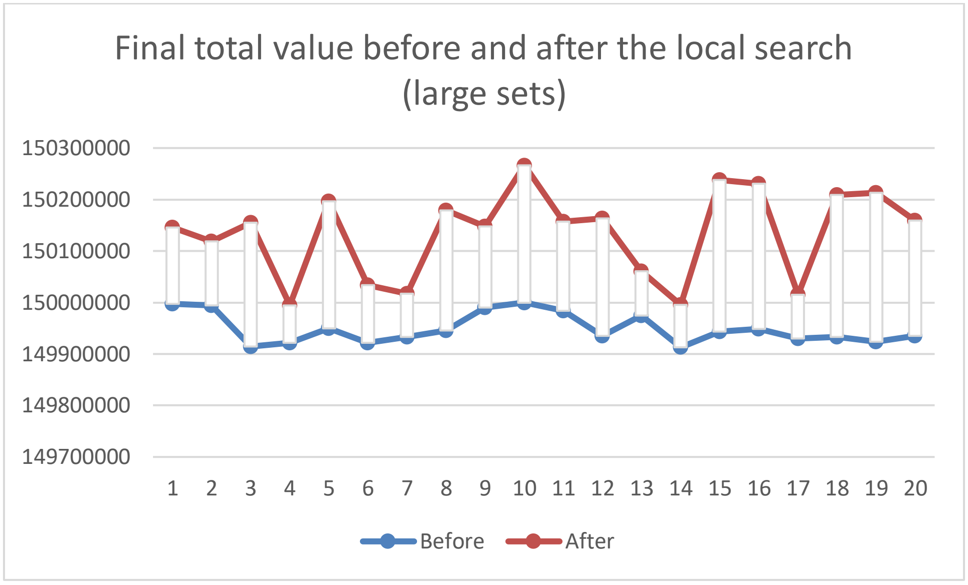


Figure 6. Final total value before and after the local search (large instances).

However, it's important to note that due to a processing time limit of 10 minutes, the program did not have sufficient opportunity to review all solutions and perform the necessary comparisons and movements. Hence, the relative improvement obtained in this group is significantly lower than in previous tests.

6. Conclusions

The results of this experiment demonstrate a consistent improvement in the final solutions after applying the local search heuristic across instances of varying sizes. On average, a relative improvement of 1.14% was achieved for small instances, 1.05% for medium instances, and 0.12% for large instances. This indicates that the local search heuristic consistently enhances solution quality, regardless of the instance size.

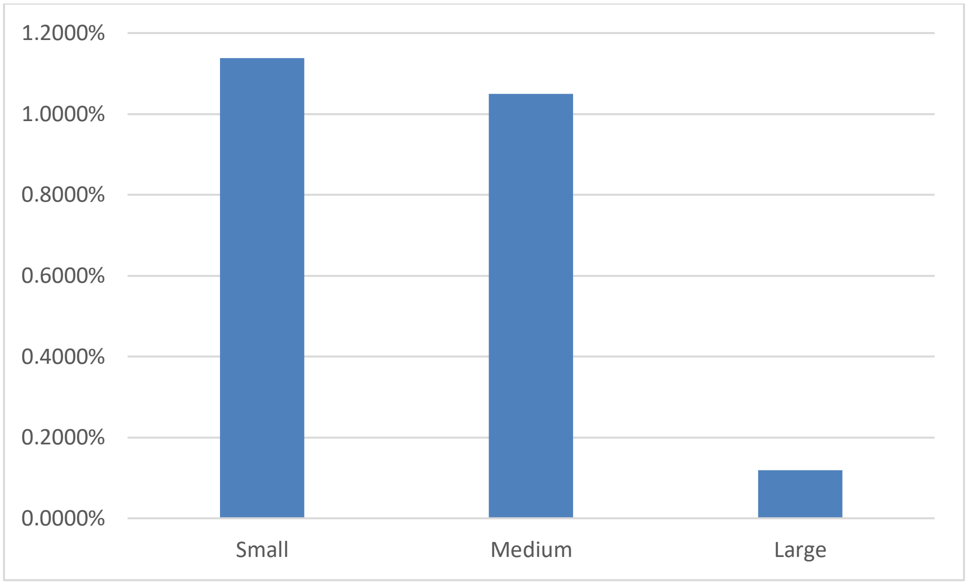


Figure 7. Final total value before and after the local search (large instances).

These findings underscore the effectiveness of the local search heuristic in refining solutions to the Multiknapsack problem. By iteratively optimizing the allocation of items across knapsacks, the local search algorithm consistently produces improved solutions compared to the initial heuristic allocation.

Moving forward, there is an opportunity for further improvement by allocating greater time and better resources to future research and experiments. By reducing constraints such as time limitations and providing access to more robust computational resources, researchers can conduct more extensive analyses and experimentation. This would enable the exploration of additional optimization techniques and the validation of results with greater statistical validity, ultimately leading to more accurate and reliable findings in the field of combinatorial optimization.

References in alphabetical order.

Gendreau, M., & Potvin, J. Y. (Eds.). (2010). *Handbook of Metaheuristics (2nd ed.).* Springer.

Lalami, M. E., Elkihel, M., Baz, D. E., & Boyer, V. (2012). A procedure-based heuristic for 0-1 Multiple Knapsack Problems. *International journal of mathematics in operational research*, *4*(3), 214. https://doi.org/10.1504/ijmor.2012.046684

Martello, S., & Toth, P. (1990). *Knapsack problems: Algorithms and computer implementations*. John Wiley & Sons.

Papadimitriou, C. H., & Steiglitz, K. (1998). Combinatorial Optimization: Algorithms and Complexity. Dover Publications.