Université Jean Monnet

MLDM Cohort 2022-2024
Advanced Algorithms
Final Project

The Knapsack Problem

Students :
Ariel GUERRA ADAMES
Mohamed MOUDJAHED
Thomas MARTINERIE
Franck SIRGUEY
Josh TRIVEDI

Professor :
Amaury HABRARD

February 3, 2025

Contents

1 Introduction

The knapsack problem can be generally defined as a problem involving the selection of a number of items with inherent weights and values, to be put into one or multiple containers or *knapsacks*, maximizing the profit of the items inside the knapsack. More formally, the simplest or classical version of this problem, also known as the 0/1 Knapsack Problem, aims at maximizing $\sum_{i=1}^{N} v_i x_i$ subject to $\sum_{i=1}^{N} w_i x_i \leq W, x_i \geq 0$, and x_i integer, where w_i and v_i correspond to the values and weights of each item x_i, and W corresponds to the weight capacity of the knapsack [5]. Since its original definition, the knapsack problem has seen a number of variations and adaptations regarding the number of knapsacks to be filled, known as the Multiple Knapsack Problem [2], or the number of features inherited by the items of interest, called the Multidimensional Knapsack Problem [1].

Considered one of the "easier" known NP-Hard problems [4], several algorithmic approaches to the knapsack problem provide efficient and practical solutions to this well-known problem in computer science and operations research, ranging from heuristic-oriented to artificial neural network-based approaches [3]. While each approach has its own strengths and weaknesses, they all offer valuable insights and solutions for solving the knapsack problem in a variety of contexts.

By taking these facts into account, we can determine that the objective of this project is to implement and analyze the performance of a number of popular algorithms with respect to different instances of the 0/1 knapsack problem. These instances include problems with a high number of samples, samples ordered according to different probability distributions, and a few multiple knapsack instances. More specifically, this study focuses on three versions of a brute force algorithm (a normal brute force approach, a backtracking approach, and a meet in the middle approach), a branch and bound algorithm, three versions of the greedy approach, two approaches to the dynamic programming algorithm, a fully-polynomial time approximation scheme, a completely naive randomized approach and a genetic programming approach.

1.1 Algorithm Descriptions

As previously mentioned, algorithms for solving the knapsack problem all involve selecting a combination of items to maximize value within a given weight constraint. There are several different algorithms that can be used to solve the knapsack problem, each with its own strengths and weaknesses. The following items attempt to briefly provide descriptions of the algorithms of interest for this project.

- The **bruteforce** algorithm involves trying all possible combinations of items and selecting the combination that yields the maximum value within the weight constraint of the knapsack. This approach is simple and easy to implement, but it can be very time-consuming and impractical for large sets of items. Its temporal complexity is often O(2^n), where n is the number of items.

- The **branch and bound** approach involves dividing the problem into smaller subproblems, or "branches," and then "bounding" each subproblem by finding an upper and lower bound on the possible solutions. This allows the algorithm to avoid exploring subproblems that are not promising, which can significantly reduce the time required to solve the problem. The branch and bound approach to the knapsack problem begins by dividing the problem into smaller subproblems, or branches. This is typically done by selecting a single item and then considering two subproblems: one in which the item is included in the solution, and one in which it is not. The algorithm then repeats this process for each subproblem, dividing it into smaller subproblems until all subproblems have been explored.

- A **greedy** approach to a given problem involves making a series of local optimal choices, with the hope of arriving at a global optimal solution. In the context of the knapsack problem, this means selecting the items with either the highest value, weight or value-to-weight ratio at each step, until the weight constraint of the knapsack is reached. Its temporal complexity is often O($NlogN$). The key advantage of the greedy approach to the knapsack problem is that it is often much faster and more efficient than other methods, such as the brute force or dynamic programming approaches.

- The **dynamic programming** approach to the knapsack problem begins by defining a set of subproblems that are smaller and easier to solve than the original problem. These subproblems are typically defined in terms of the number of items considered and the weight of the knapsack. In the case of our implementation of interest, this approach uses a matrix to store calculations. Next, the algorithm solves each subproblem individually, using a recursive approach. This involves solving the subproblems that are smaller and simpler first, and then using those solutions to solve the larger and more complex subproblems. Calculations of value of objects already performed can be found by calling items in a matrix M[n, W] (n: number of object, W: capacity of the knapsack). Its temporal complexity is often O(nW). Another version of this approach knwon as the **top down** approach, reduces the number of recursive calls if possible.

- The **Fully Polynomial Time Approximation Scheme (FPTAS)** is a variant of the dynamic programming algorithm. It begins by dividing the range of possible values for the items into a finite number of discrete intervals, or "buckets." Each bucket represents a range of values, and the items are sorted into the appropriate bucket based on their value. Next, the algorithm applies the dynamic programming approach, using the buckets as a way to divide the problem into smaller subproblems. This allows the algorithm to avoid considering all possible combinations of items, which can be very time-consuming and impractical for large sets of items. Instead, the algorithm only considers the items in each bucket, which significantly reduces the time complexity of the problem. The key advantage of the FPTAS for the knapsack problem is that it is a fully polynomial time algorithm, which means that it can solve the problem in a time complexity that is polynomial in the size of the input, or n and ϵ more precisely.

- In a **randomized** approach to the knapsack problem, a random solution is generated and then improved upon iteratively until an acceptable solution is found. If the change in value is positive, the new solution is accepted, and the process is repeated. If the change in value is negative, the new solution is rejected, and the process is repeated with the original solution.

- The **genetic** approach of the knapsack problem involves representing the potential solutions to the problem as a population of "chromosomes," which encode different combinations of items. The algorithm then applies a set of evolutionary operators, such as selection, crossover, and mutation, to evolve the population over time and find high-quality solutions to the problem.

- The **meet-in-the-middle** approach to the knapsack problem begins by dividing the items into two sets, typically of equal size, and then solving the problem separately for each set. The solutions to the two sets are then combined to obtain the solution to the original problem. This is typically done by creating a "lookup table" for each set, which maps the weight of the knapsack to the maximum value that can be obtained for that weight. The algorithm then searches the lookup tables to find the combination of items from the two sets that yields the maximum value within the weight constraint of the knapsack. Its temporal complexity is often $O(n2^{n/2})$.

From here on, this report is structured as follows: Section 2 presents and describes the methodology of the present study, including but not being limited to the computational tools and experimental procedures used in this project. Section 3 displays the results product of the previously explained methodology, and Section 4 aims at drawing conclusions from such results. Finally, the Checklist Discussion section presents the completed tasks assigned by the project description, as instructed.

2 Materials and Methods

This section aims at providing a complete description of materials and methodologies implemented throughout the course of the project. The first part provides an in-depth look at the experimental setup, and overall workflow of the team. The second and third parts describe the data generated and extracted for experimentation purposes, and the fourth part describes the experiments performed with the aforementioned elements.

2.1 Experimental Setup

In order to obtain homogeneous results in terms of the time performance of every algorithm, all experiments enlisted in this report were performed on the same computer. This machine is a 2020 13-inch Macbook Pro with a 1.4 GHz Quad-Core Intel Core i5 processor, and 8 GB of LPDDR3 RAM. All of the scripts were developed on the Python programming language, version 3.9, and experiments were performed using the PyCharm IDE, version 2022.3, Community Edition.

Collaborative work was done using Git as version control, specifically through a repository hosted on GitHub. For transparency and repeatability purposes, this repository will be publicly available on the date of the defense. A copy of the repository up to the final date will be equally added on the compressed file to be turned over to the instructors.

Work was divided in a parallel fashion. Each team member worked individually in their IDE of choice, on their algorithms of choice, and verified functioning with the agreed-upon data. Once the algorithm was finished and tested by the individual, regularized testing could be performed on the aforementioned machine.

2.2 Problem Generator

With the objective of observing any possible impact that the structure of the data could have on the accuracy or execution of each of the algorithms, three problem generators were developed, based on three basic probabilistic distributions: the continuous uniform probability distribution, the Gaussian probability distribution, and the triangular probability distribution. All three problem generators take a series of parameters related to their respective distribution, and return an instance of the 0/1 knapsack problem containing the list of values, weights, capacity of the knapsack, and optimal solution if requested. The generators have a default random seed to guarantee reproducibility of the experiments, and the option to calculate the optimal solution based on the brute-force algorithm. It is worth noting that because of the exponential time-complexity of the brute-force algorithm, generation of instances with the optimal solution is limited to small knapsack instances.

The generator based on the uniform distribution takes a series of parameters such as minima and maxima of the profit and weight of the items, as well as the desired capacity of the knapsack and the number of items to be generated. An example composed of 1000 samples is shown in figure 1 below, along with a histogram describing its probability distribution.

The generator based on the Gaussian distribution takes a series of parameters such as the mean and standard deviation of the profit and weight of the items, as well as the desired capacity of the knapsack and the number of items to be generated. An example

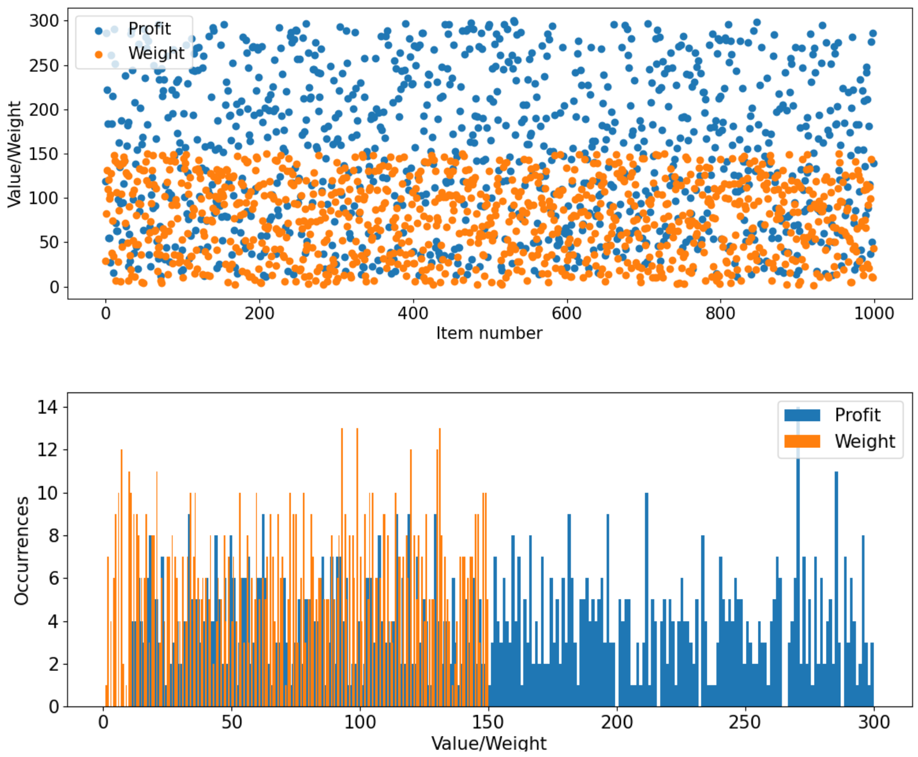

Figure 1: Uniformly-distributed instance of the 0/1 Knapsack Problem.

composed of 1000 samples is shown in figure 2, along with a histogram describing its probability distribution.

The generator based on the triangular distribution takes a series of parameters such as the minima, maxima and modes of the profit and weight of the items, as well as the desired capacity of the knapsack and the number of items to be generated. An example composed of 1000 samples is shown in figure 3. This distribution was selected because of the suspicion that greedy approaches can perform poorly if the distribution of the weight and profit values oppose each other.

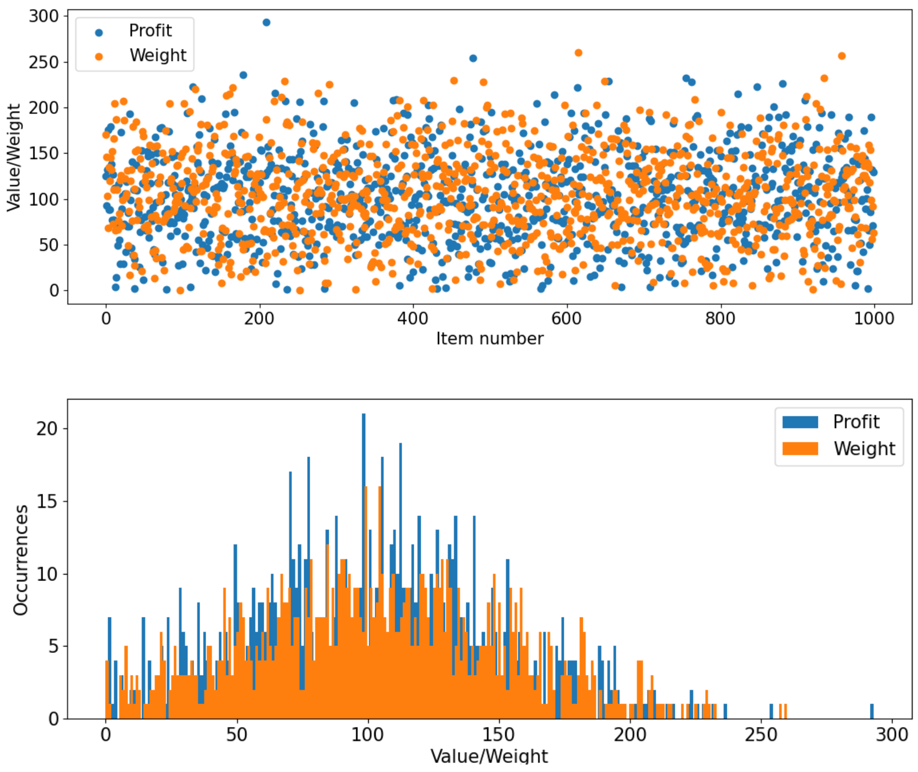

Figure 2: Gaussian-distributed instance of the 0/1 Knapsack Problem.

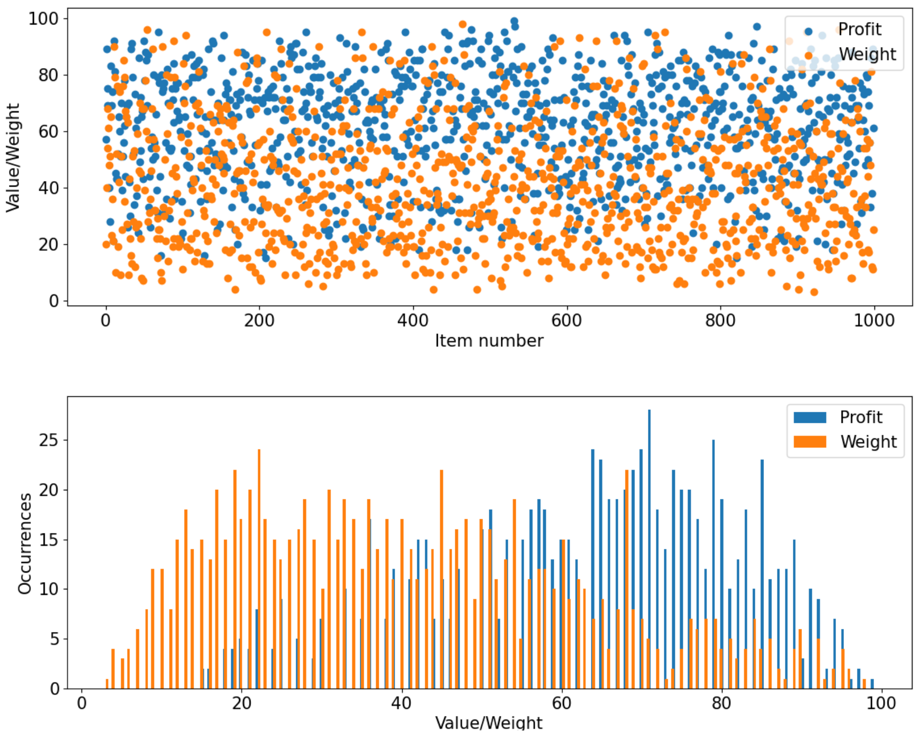

Figure 3: Triangular-distributed instance of the 0/1 Knapsack Problem.

2.3 External Datasets

Apart from the previously specified problem generator, we decided to evaluate the results on two external datasets: the low-dimensional 0/1 knapsack problem dataset and the multi-knapsack problem dataset, both given in the project description. The first of these datasets contains 10 project instances, with different numbers of items and different probability distributions shown in figure 4. Some instances such as the first and second one appear to be uniformly distributed, so we expect to obtain generally accurate solutions across all algorithms. Some other instances such as the third and fourth instances appear to be uniformly oppositely distributed, with the added constraint of only having four sample, possibly posing problems for greedy approaches. Instance five introduces decimal values, which depending on the rounding algorithm used may impact solution accuracy. Instances six and seven appear to introduce a skew on the distribution of the data, while instance eight introduces higher values with a tendency towards the maximum value. Instance nine and ten once again appear to be quite uniformly distributed instances, with the added constraint of a small number of values in the ninth instance.

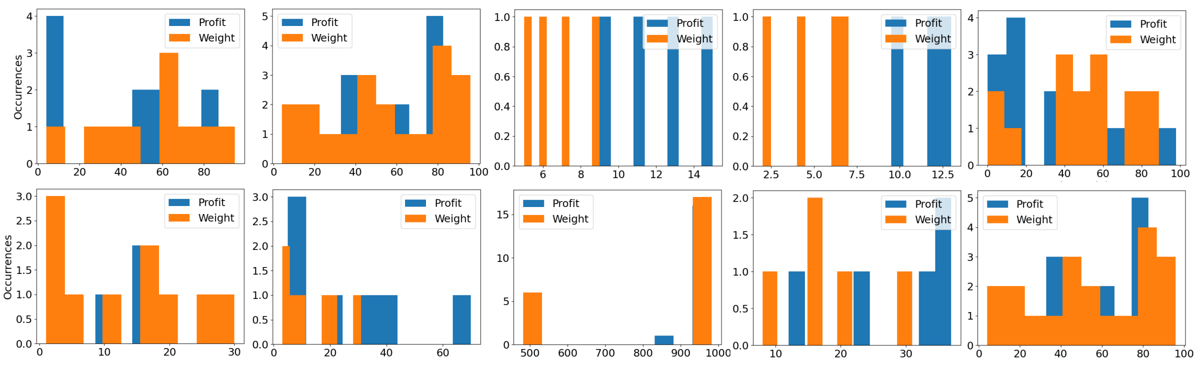

Figure 4: Histograms of all instances in the provided dataset.

Instance / Property	f1	f2	f3	f4	f5	f6	f7	f8	f9	f10
n of samples	10	20	4	4	15	10	7	23	5	20
profit mean	41.2	54.25	12	10.25	37.1333	10.5	26.857142	839.52173	28.4	54.3
weight mean	53.9	54.9	6.75	4.75	48.86666	13	13.2857	844.6956	18.2	54.9
profit std	30.78	24.5903	2.236	2.680951	30.146015	7.04627	21.76122	212.2762	9.393614	24.590851
weight std	26.2885	29.7958	1.47901	1.92028	26.90196	10	9.61716	213.5113	7.52063	29.7958
profit skew	0.0873	-0.29548	0	-0.6568	0.511743	-0.16207	0.8283	-1.05138	-0.801976	-0.30157
weight skew	-0.3824	-0.27088	0.43465	-0.27803	-0.36168	0.1866	0.60016	-1.0855	0.473604	-0.27088
profit kurtosis	-1.434	-1.05508	-1.36	-1.098979	-0.9758	-1.609252	-0.47511	-0.852715	-0.894549	-1.052801
weight kurtosis	-0.74051	-1.26049	-1.15428	-1.4266	-0.730976	-1.32235	-0.94377	-0.816026	-0.6377	-1.26049
capacity	269	878	20	11	375	60	50	10000	80	879
optimal	295	1024	35	23	481.0694	52	107	9767	130	1025

Table 1: Main statistical properties of the low-dimensional dataset.

The second dataset contains six instances of the multiple knapsack problem, out of which only three contain the solution strings. All three instances consist of a list of profits and weights of items to be placed in two knapsacks, with a list of the capacities of each knapsack, and the optimal solution vectors. This would allow for the calculation of solution accuracy of the algorithms adapted to this task.

2.4 Algorithms Implemented

As mentioned on Section 1.1, a number of popular algorithms were implemented to be experimented on based on different instances of the 0/1 Knapsack Problem. In the subsections to follow, a brief explanation of each implementation is given.

2.4.1 Brute Force Algorithms

Given the option, the group decided to implement three versions of brute force algorithms, based on popular implementations of this algorithm. The first, and simplest brute force approach, iterates over all the possible combinations. This algorithm takes a list of profits, weights, and capacity of the knapsack, and codes them into a one-hot coded vector. This vector is then used to calculate the value of all the objects in the knapsack. As it iterates exhaustively over all combinations, this algorithm should always find the optimal solution, at the expense of an exponential time complexity.

A second approach based on backtracking was also implemented. This approach takes our list of profits and weights, and recursively sends it to a knapsack function which compares the value of each profit and notes its index if the weight doesn't exceed the capacity. When current weight is less than our max Weight, we keep the item, and repeat the process till we find the weights (sum) greater than the max weight. We backtrack after this step is reached. We initialize two sets (knapsacks) during our problem, one that stores our calculation at recent step, and the other that stores max weight, current weights, values and index. These are recursively changed based on our current state and when our algorithm reaches its max capacity, If according to our first set calculation at index 0 (weight does not exceed max weight), we keep the item, and proceed till our weights exceed the max weight. We backtrack to the last index of our set of weights, repeat the calculations and return our knapsack.

A third approach based on the meet in the middle algorithm was also proposed. This algorithm takes the set of items to be put in the knapsack and divides it in two subsets A and B. We calculate then the object combinations for each subset and the sum of these combinations. Then we look for the subset B with the highiest returning value such that its weight fill the capacity W of the bag.

2.4.2 Branch and Bound Algorithm

The branch and bound algorithm takes our list of profits and weights in the order they are generated or given by the database. This is a recursive algorithm that takes each object $k \leq n$, and then look if putting the bag is a good idea by creating two "branches", one that will include the object and another that won't include it. Then, inside each recursive branch, we look at the $k+1$ object in the same way and we create branches until the next object does not fill into the bag).

Additionally, each time we create a new branch, we evaluate its "potential", aka the estimated sum of the best profits that we will get by going further and creating new branches. This way the algorithm will realize that he doesn't need to go any further in a specific branch and thus it will simply cut it, by ignoring it. The goal in the algorithm is to find the branch with the best potential, and each time we create a branch that has less potential than the best one we got so far, then we cut it.

Once the bag is almost full, or when we ended being on the last object in the database, then a solution is found, and we explore the remaining branches that we didn't explored yet. At the end of the algorithm, the best "branching path" gives us our solution and its profit.

This method is mandatory because otherwise this algorithm will basically be in the same time complexity range as the bruteforce algorithm. But we can expect that its complexity will vary depending on how the data is distributed. For example if the algorithm finds a branch with an incredible potential at the very beginning, then there is a high chance that it will cut all its branches left. On the contrary, if the best potential is founded at the very end, it means that they're is a chance that the algorithms went into a lot of unnecessary branches for nothing.

2.4.3 Greedy Algorithms

As instructed, three versions of a greedy algorithm were proposed. They correspond to a value-oriented greedy algorithm, a weight-oriented greedy algorithm, and a fractional greedy algorithm. All three algorithms take in a list of profits and weights of items, and the capacity of the knapsack.

The first of the three algorithms, sorts the items on a value-wise decreasing order, using a bubble sorting algorithm because of its low time complexity. Once the items have been sorted, it proceeds to add items to the knapsack one by one, checking if the weight limit has already been reached. Once it has been reached, the algorithm calculates the total value of the items in the knapsack, and returns it along with the one-hot coded solution.

The weight-oriented version of this algorithm works in a similar fashion, but ordering items on a weight-wise decreasing order. The fractional version calculates a value-to-weight ratio for every item, hence the name *fractional*, and orders the items in a ratio-wise descending order. From there on, it operates in the same way as the preceding versions of the algorithm.

2.4.4 Dynamic Programming Algorithm

The implemented dynamic programming approach, follows a series of steps enumerated as follows:

1. We read the list of profits Lv, the list of weights Lw, and the capacity W of the knapsack.
2. We create a matrix M, with its values initialized to the value 0
3. To change a value inside the matrix there are two cases :
    (a) If the weight Lw[i] > W, then we can't add anything else, we get back its upper value M[i-1, w]
    (b) Else, the value will be the maximum between its upper one M[i-1, w] and the value M[i-1, w-Lw[i]] + Lv[i], which correspond to the value with the highest capacity that has already been calculated, plus the value we need to add.

The **top down approach** utilizes a recursive algorithm that takes the same rules than the dynamic approach. However, the recursive calls are only done when we need a value that wasn't already calculated in the matrix.

2.4.5 Fully-Polynomial Time Approximation Scheme

Our FPTAS implementation takes as inputs a list of profits L_v, a list of weights L_w, the capacity of the knapsack W, and a constant ε. First, we search for the maximum value in L_v. We then compute an adjustment factor as $k = (max(L_v) \times \varepsilon)/n$, where n is the number of items to be arranged in the knapsack. We then adjust all the profits dividing them by the new adjustment factor k, storing them in a new list L'_v. Once this new list has been obtained, we use it to perform the top down approach of dynamic programming, returning the maximum profit of the knapsack. The value of this profit equals at least 1-ε times the correct value and at most 1+ε times the correct value. To adjust the precision of the algorithm it will take the original profits back from their original list thanks to the returned indexes.

2.4.6 Randomized Algorithm

For this algorithm we create a list of n random numbers inside, from 0 to n-1, that never appear twice. They will represent the indexes of our objects, so that this random list is a random order to pick them. Then we recursively take them in the order until the bag is full (or until the next object doesn't fill the bag) to get a random solution. To increase the chances to have a better result, the number of iterations must be increased, and we take the solution with the highest profit.

2.4.7 Genetic Algorithm

Our implementation of a genetic algorithm starts by generating a population of candidate solutions to the problem, where each candidate is represented as a list of 0s and 1s, indicating whether each item should be included in the knapsack or not. It then obtains a "fitness" for each candidate solution by calculating its value and weight, and determine whether it is a feasible solution to the problem, by evaluating whether its weight is less than or equal to the knapsack's maximum capacity, and by adding its value to the profit. It then selects the fittest candidates from the population. After that, it uses the genetic operators crossover and mutation to produce new candidate solutions from the selected fittest solutions. Crossover involves combining the genetic information from two parent solutions to produce a new offspring solution, while mutation involves randomly modifying the genetic information of a solution to produce a new, slightly different candidate. Afterwards, it repeats the previous steps for a number of generations, allowing the population to evolve and improve over time. The algorithm stops when a pre-determined number of generations have been reached.

2.5 Multiple Knapsack Problem

To explore the performance of some of the algorithms, we decided to adapt them to work with multiple knapsack instances. These algorithms were the simplest version of the brute force algorithm, and the three greedy algorithms.

The brute force algorithm for example, works in a very similar fashion as its single-knapsacked counterpart, with the added complexity that conveys calculating twice the number of combinations given the weight capacity constraint of the each knapsack. It starts by generating all possible combinations of items that can be placed in the knapsacks. At each step, it chooses whether to include or exclude each item in the current combination. For each combination, it calculates the total value and weight of the items, and determines whether it is a feasible solution (i.e. whether the weight of each item does not exceed the capacity of its corresponding knapsack). At the end, it selects the combination with the maximum total value among all feasible solutions.

The multi-knapsack version of the greedy approaches also work in a similar fashion as their single-knapsacked counterpart. The difference lies in the fact that on the multiple versions, the "filling" part of the algorithm attempts to fill knapsacks in a serialized manner. In other words, after ordering the items in decreasing order, the algorithm places items in one knapsack until it reaches its weight capacity, and then moves on to the second knapsack, until they are both "full". While based on heuristics, the fact that this algorithm must now perform a k-number of comparisons every time there is m

2.6 Data Extraction

In order to normalize the workflow among algorithms with the objective of simplifying experimentation. A series of extraction functions were developed around the structure of the data of the low-dimensional and multiple knapsack datasets. These functions read the text file containing the lists of items, values, and optimal values in the knapsacks, and returns them in separate lists to be interpreted by each individual algorithm.

In addition to these functions, a directory containing one-hot coded optimal solution vectors based on the backtracking algorithm was generated from the low-dimensional dataset. This was done to evaluate the quality of the solutions provided by each of the algorithm.

2.7 Experiments Performed

With the objective of extensively testing each of the implemented approaches, trying to obtain as much information related to their performance as possible, a number of experiments and metrics were devised and implemented. However, given the exponential nature of the brute force approaches, the scale of time performance-related experiments was orders of magnitude different with their non-brute force counterparts. Generally, it was sought to measure three specific metrics related to their performance:

- **The execution time** to be evaluated over a range of number of items in the knapsack. Depending on the family of algorithm, the unit of measurement could be either seconds or milliseconds.

- **The solution accuracy** to be evaluated as follows:

$$Accuracy = \frac{\sum_{i=1}^{N} v_i x_i}{v_o}$$

  where the numerator corresponds to the sum of selected values by the algorithm, and v_o corresponds to the optimal profit given by the instance or generator.

- **The solution quality** to be evaluated as follows:

$$Quality = \frac{lev(x_o, x_i)}{N}$$

where $lev(x_i, x_o)$ corresponds to the Levenshtein distance between the optimal solution x_o and the obtained solution x_i coded as one-hot lists, in relation to the number N of items each instance.

Each of the performed experiments with regards to the origin of the instances is to be explained in the following subsections.

2.7.1 Problem Generator

As it is described in Section 2.2, the problem generator was used to create a number of instances of varying sizes and probabilistic distributions. Once again, given the expected time-performance of each of these approaches, these experiments were implemented.

- For the **brute force** family of algorithms, a set of experiments which measures the execution time of the algorithms among a range of item values which iterates increasingly from 3 items to 30, with a granularity of one item.
- For the **Greedy, Dynamic, Polynomial, Randomized, and Genetic** families of algorithms, a set of experiments which measures the execution time of the algorithms among a range of item values which iterates increasingly from 10 to 1010 items, with a granularity of 100 items.
- For the **Greedy, Polynomial, Randomized, and Genetic** families of algorithms, a set of experiments which measures the solution accuracy algorithms among a range of item values which iterates increasingly from 3 to 20 items, with a granularity of one item, and calculates the average accuracy depending on the probabilistic distribution. The size of these experiments was limited to 20, as the calculation of accuracies above this level required the execution of the brute force algorithm to obtain such responses.

2.7.2 Datasets

As it is described in Section 2.3, problem instances were also extracted from the datasets provided by the project assignment. For the low-dimensional dataset, it was considered that as its 10 instances contain the optimal value assignments for each of the instances, and we obtained the optimal solution vector based on such values and the bruteforce algorithm, we devised the following experiments to be performed:

- For **all** algorithms, the execution time for each individual instance of the dataset was measured.
- For the **Branch and Bound, Value, Weight and Fractional Greedy, Polynomial, Randomized and Genetic** family of algorithms, the solution accuracy was measured.
- For the **Branch and Bound, Value, Weight and Fractional Greedy, Polynomial, Randomized and Genetic** family of algorithms, the solution quality based on the edit distance ratio was measured.

For the multiple knapsack dataset, as only the first, fifth and sixth instances contain information related to the optimal solutions, the following experiments were devised:

- For the **Brute Force and Greedy** families of algorithms, the execution time for each individual instance of the dataset was measured.
- For the **Greedy** family of algorithms, the solution accuracy and execution times were measured.

Results obtained from the previously described experiments are displayed in the following section.

3 Results

This section presents the findings of the project, including the performance and characteristics of all the algorithms mentioned in the introduction, and any insights that were gained through the study. To evaluate the performance of the algorithms, we performed a series of experiments on a set of test instances, and generated problems using different combinations of input parameters.

3.1 Time Performance

This subsection describes the computational time required by each algorithm to find solutions for the test instances. This information is useful for assessing the efficiency and scalability of each algorithm, and for comparing its performance with other algorithms for the same problem.

3.1.1 Brute force approaches

Because of its expected exponential time complexity, we measured the time performance of all brute force approaches (brute force, backtracking and meet in the middle) separately and differently from the rest of the algorithms. First, we generated 27 different instances of ascending numbers of items using the uniformly distributed problem generator. The time it took the algorithm to find an optimal solution in each iteration was recorded and plotted as its shown on Figure 5.

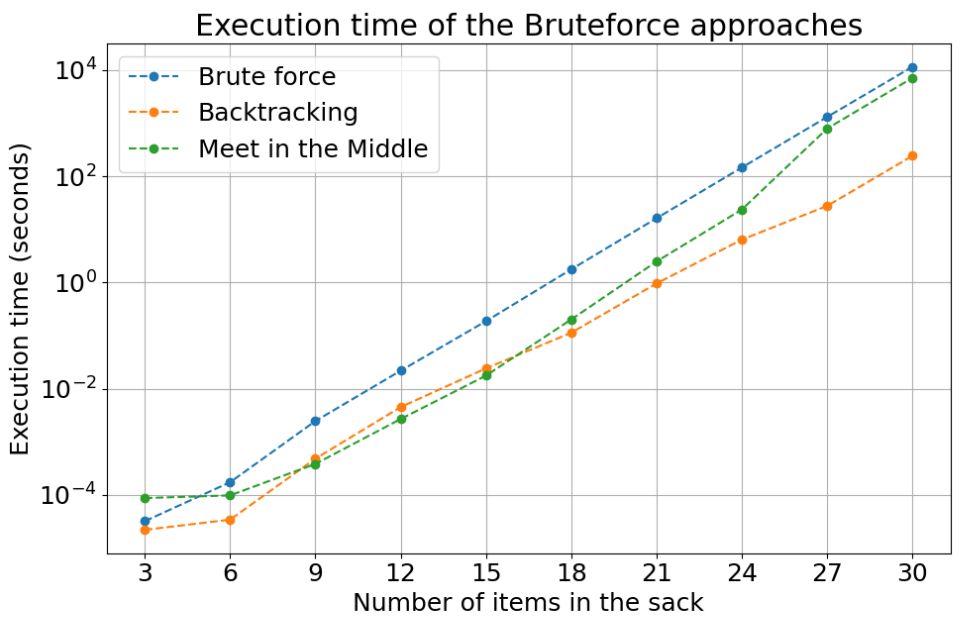

Figure 5: Execution time of brute force approaches

These time performance results allow us to confirm empirically that the brute force algorithms have an exponential time complexity, as the elapsed time increases exponentially with the size of the test instances. It also confirms that both the backtracking and meet in the middle algorithms slightly outperform the simplest brute force approach because of their slightly optimized search methods, reducing the number of iterations/recursive calls that need to be made.

From here, we proceeded to measure the time performance of each low dimensional dataset instance for every brute force algorithm. These results are displayed on Table 2.

Instance	**Execution time of algorithms (ms)**		
	Bruteforce	**Backtracking**	**MiM**
f1	4.1298	0.68402	0.50306
f2	7724.5411	1299.7241	897.95207
f3	0.10085	0.0441	0.110149
f4	0.09083	0.041007	0.12207
f5	206.9239	23.05507	16.375064
f6	4.8999	0.61511	0.579833
f7	0.39196	0.17619	0.16903
f8	69545.2837	5848.47831	8848.0949
f9	0.180006	0.0629425	0.157833
f10	7659.6701	1312.37483	852.9129

Table 2: Time performance results in milliseconds of brute force approaches when testing on the low-dimensional dataset.

These results allow us to see a clear difference in time performance between the simplest brute force approach and its more optimal counterparts. For example, the time difference between the meet in the middle algorithm and the brute force algorithm for the f2 instance, which is composed of 20 items, is almost one order of magnitude.

3.1.2 All other approaches

As the rest of the algorithms are expected to perform significantly better than the brute force approaches in time performance terms, we decided to test them separately. As previously explained, these algorithms were first tested using a problem generator, but unlike with the brute force approaches, we generated uniformly-distributed instances containing between 10 and 1010 items. This would allow us mostly to test the scalability of each algorithm when faced with large-dimensional problems. The results of these tests were recorded and are shown in Figure 6.

These graphical results allow us to empirically confirm the expected behaviour of each of these algorithms, and draw a couple of observations from the data. First and most obviously, it's evident that the genetic algorithm's time performance is constant with respect to the number of items in the instance, because of how it estimates the best possible solution. Instead, the genetic algorithm's time complexity depends on the number of individuals used to populate every generation. This time performance analysis was repeated for an iteratively higher number of individuals, and a constant number of items. The results of this experiment is shown in Figure 7. It is also evident that, as expected, the greedy algorithms outperformed all other algorithms by large, with a slight advantage for the value and weight-oriented versions because of the lack of the ratio calculation. The randomized approach follows as it was programmed in a naive fashion, with the dynamic and polynomial approaches being among the worst performing time-wise.

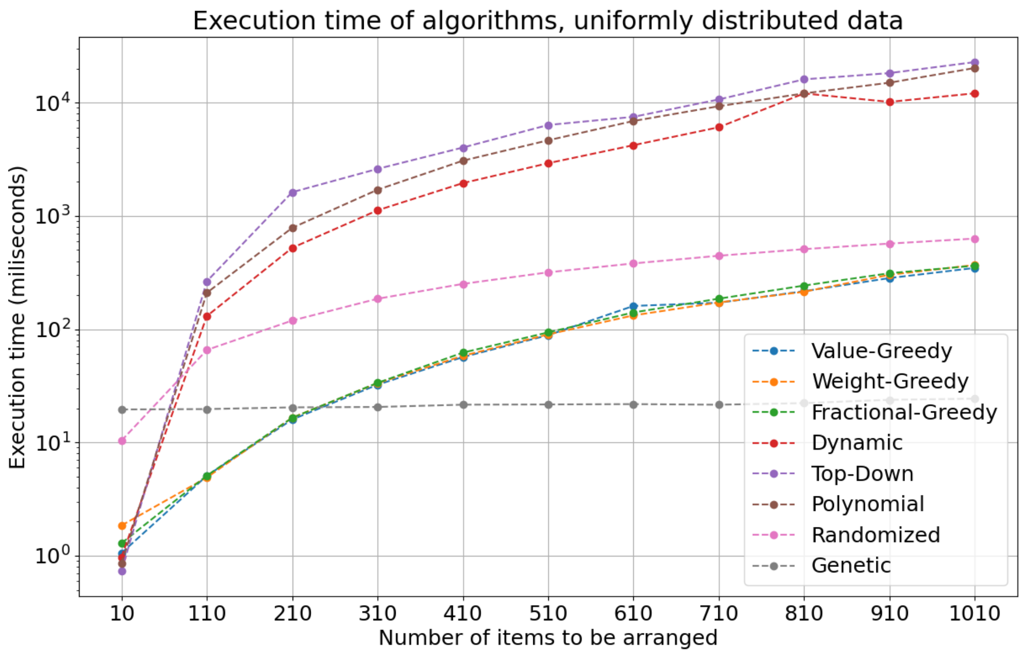

Figure 6: Time performance of all non-brute force algorithms when testing with the problem generator, in milliseconds.

We also repeated testing on the low-dimensional dataset for all other algorithms. These results are shown in Table 3.

Instance	Execution time of algorithms (ms)						
	BaB	V-Greedy	W-Greedy	F-Greedy	Poly	Randomized	Genetic
f1	0.06294	110.5637	23.873	0.47588	0.23698	0.31876	1.14011
f2	0.36501	210.535	23.42605	0.69904	0.34189	0.40984	13.91005
f3	0.02193	66.3156	19.8421	0.3719	0.17619	0.22292	0.20289
f4	0.02717	53.431749	21.95501	0.32281	0.1378	0.13899	0.250816
f5	0.1111	139.0089	23.54311	0.61893	0.25701	0.61488	3.31497
f6	0.08225	112.4358	21.73185	0.53215	0.1831	0.211	1.03092
f7	0.03385	96.4601	19.20604	0.59199	0.33783	0.17023	0.34999
f8	0.28276	207.8049	23.42391	0.985145	0.294923	0.34689	23.0081
f9	0.02503	72.17311	19.429206	0.62799	0.357866	0.275135	0.266075
f10	0.22387	200.692892	20.42078	0.696182	0.338077	0.4048347	14.7159

Table 3: Time performance results of all other approaches when testing on the low-dimensional dataset, in milliseconds.

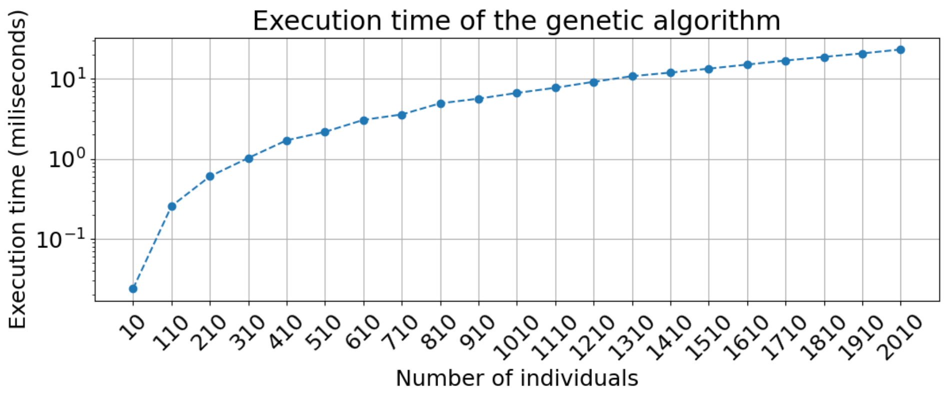

Figure 7: Time performance of the genetic algorithm when testing with the problem generator, in milliseconds.

3.2 Accuracy of the Algorithms

Experiments to assess the optimality of the solutions of the implemented algorithms were also performed using both our problem generator, and the low-dimensional dataset. However, because the brute force approaches are always expected to find optimal solutions, they are not tested for this purpose. Instead, the simplest brute force approach is used to obtain the optimal solution vector for the generated instances. In the following subsections, the results of experiments to assist the solution accuracy and quality are presented.

3.2.1 Solution Accuracy

First, solution accuracies were calculated as specified on Section 2.7 for every instance of the low-dimensional algorithm. The results of these tests are shown both in Figure 8 and Table 9. It is worth mentioning that because both dynamic approaches achieved accuracies of 100% for every instance that we tested them upon, they are not included in Figure 8.

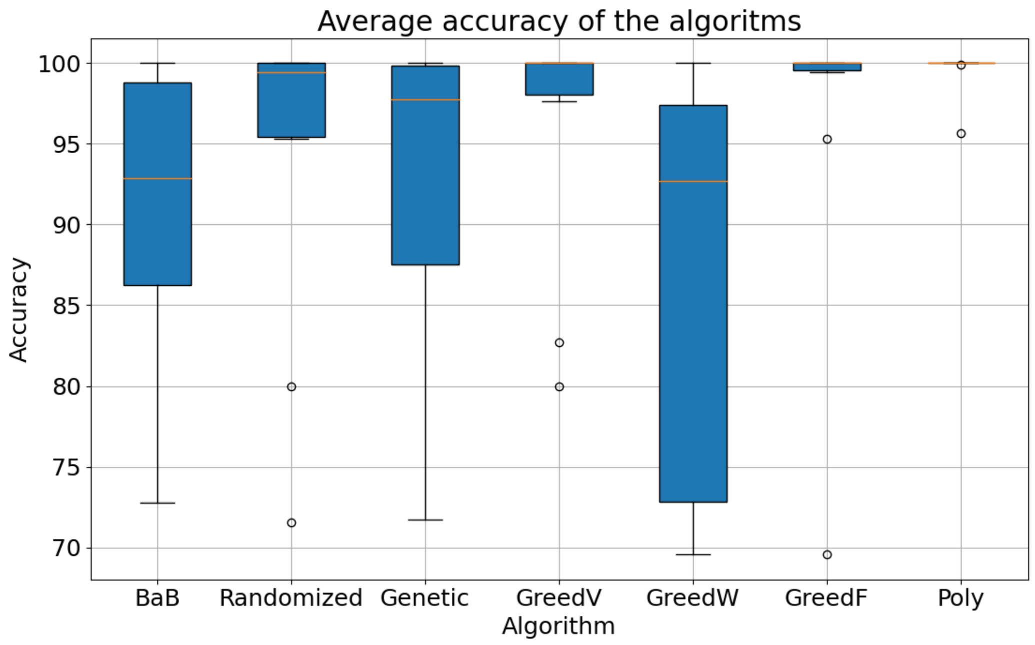

Figure 8: Accuracies of non-brute force algorithms (%).

Upon inspection of these results, it is immediately evident that the dynamic and FPTAS approaches outperform all other approaches, and that the weight-greedy algorithm performs the poorest, as expected. We can also observe an overall excellent performance for the fractional greedy approach, with the exception of the fourth instance, where the data pertaining to the profits and weights is maximally uncorrelated, and there exists only four samples. In the case of the genetic algorithm, its accuracy could possibly be improved if the number of individuals in each generation is increased, but it was deemed unnecessary as with the current number of individuals (10), the algorithm performs adequately.

Tests using the problem generator were divided into the three probability distributions previously described statistical distributions. In a similar fashion as what has been done

Instance	Accuracy of Algorithms (%)							
	BaB	Randomized	Genetic	V-Greedy	W-Greedy	F-Greedy	Dynamic	Polynomial
f1	85.08474	99.66101	99.322	97.6271	72.5423	100	100	100
f2	81.34765	100	84.375	100	92.6757	100	100	100
f3	94.2857	80	100	80	100	100	100	100
f4	100	95.6521	100	100	69.56521	69.565217	100	95.6521
f5	72.7545	99.15409	71.7152	99.154	71.5073	100	100	100
f6	96.1538	100	96.15384	82.6923	96.15384	100	100	100
f7	89.7196	95.3271	86.91588	100	73.8317	95.327102	100	100
f8	99.6723	99.92833	99.35497	99.9795	97.8191	99.8566	100	99.89
f9	100	71.53846	100	100	100	100	100	100
f10	91.4146	100	89.26829	100	92.68292	99.41463	100	100

Table 4: Accuracy results of certain approaches when testing on the low-dimensional dataset.

for the time-performance experiments, a number of problem instances were generated with numbers of items ranging from 3 to 20, and their optimal solution was calculated. These problem instances were used to test each of the algorithms, calculating the solution accuracy on each iteration. The average accuracy results for these experiments are enlisted in Table 5.

Distribution	Accuracy of Algorithms (%)						
	V-Greedy	W-Greedy	F-Greedy	Poly	Randomized	Genetic	Dynamic
Uniform	94.54	90.65	97.7	100	95	85.19683	100
Gaussian	92.01	92.34	98.2	100	96.01	60.77161	100
Triangular	93.87	94.83	98.38	100	95.73	90.49434	100

Table 5: Accuracy results of certain approaches when testing on different instances generated by the problem generator.

From these results, a number of relevant observations can be made. In the first place, and in accordance with what has been observed on the previous experiment, the statistical distribution of the data does have an effect on the accuracy of the algorithms. For example, on minimally correlated instances generated with the triangular-distribution problem generator, the two versions of the greedy algorithm which depended on sorting either profits or weights performed much more poorly than their fractional counterpart, which takes into account both factors. A similar behaviour was observed for the Gaussian-distribution problem generator, which resulted in an extremely bad performance for the genetic algorithm, and a sub-par performance for the value and weight-oriented greedy algorithms.

3.2.2 Solution Quality

As a final part of the experimentation for the single-knapsack instances, a solution quality metric based on the Levenshtein distance was retrieved. As previously explained, this metric was empirically calculated by dividing the edit distance between the obtained solution string and the optimal solution string by the number of items in every instance. This metric was calculated for every instance of the low-dimensional dataset, and the results are shown in Figure 9 and Table 6.

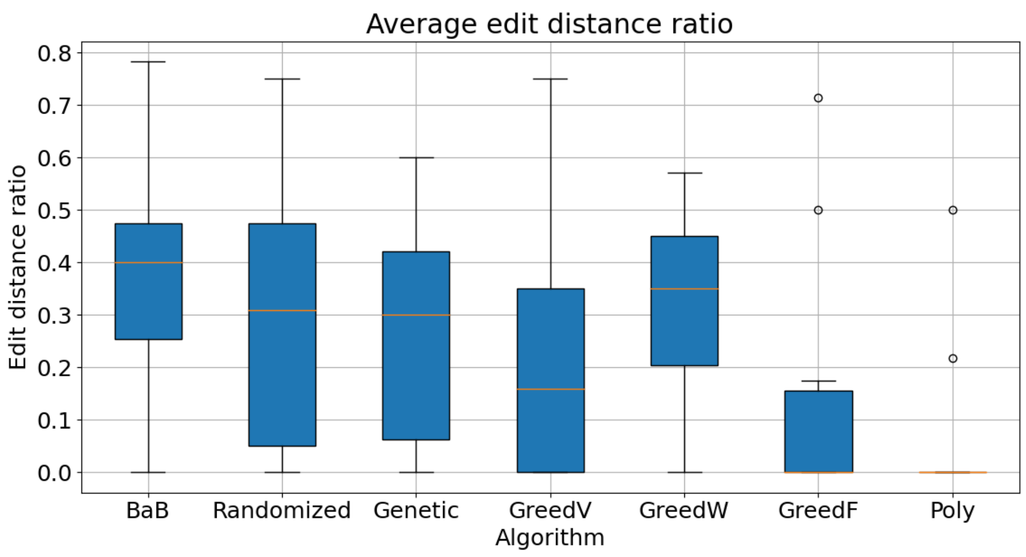

Figure 9: Edit distance ratio of non-brute force algorithms.

Instance	Edit distance ratio of algorithms						
	BaB	**Randomized**	**Genetic**	**V-Greedy**	**W-Greedy**	**F-Greedy**	**Polynomial**
f1	0.4	0.2	0.3	0.4	0.4	0	0
f2	0.4	0	0.25	0	0.2	0	0
f3	0.5	0.75	0	0.75	0	0	0
f4	0	0.5	0	0	0.5	0.5	0.5
f5	0.2666	0	0.4	0	0.4666	0	0
f6	0.4	0	0.6	0.6	0.4	0	0
f7	0.5714	0.7142	0.4285	0.1428	0.5714	0.7142	0
f8	0.7826	0.2173	0.5217	0.1739	0.2173	0.1739	0.2173
f9	0	0.4	0	0	0	0	0
f10	0.25	0	0.3	0	0.3	0.1	**0**

Table 6: Solution quality measure obtained from algorithms of interest.

While these results seem to be overall in accordance with the behaviour observed in the solution quality subsection, they offer slightly changed set of observations. For example, while the fractional version of the greedy algorithm appeared to have an accuracy comparable to that of the FPTAS algorithm, it's solution quality appears to be more in par with that observed on the other greedy approaches. This could be a decision-making factor when choosing this algorithm on real-world problems.

3.3 Multiple Knapsack Results

Finally, as an expansion to the previously obtained insight, a small set of experiments related to the adaptation of two of the algorithms to the multiple knapsack problem were performed. Using three problem instances of a multiple-knapsack problem dataset which contained the optimal values and one-hot coded solutions, the three approaches to the greedy algorithm were measured. These results are shown on Table 7.

Instance \ **Algorithm**	**Accuracy (%)**		
	Value	**Weight**	**Fractional**
KP1	91.61	92.61	97.71
KP5	100	83.87	86.24
KP6	91.78	92.71	92.71

Table 7: Accuracy results of greedy approaches when testing on the multiple knapsack problem.

This small set of experiments allows us to perform a number of observations. In the first place, it is evident that both the first and the sixth instances were particularly difficult for these versions of the greedy algorithm. This is probably due to the fact that in both instances, the items' profits and weights appear to be oppositely distributed. The fifth instance on the other hand seems to privilege the value-wise version, probably due to the fact that in this instance, values seem to be skewed. However, because of the small number of samples, further testing needs to be performed to verify these observations.

4 Conclusions

Having finalized all implementations and observations stemming from the experiments, we have reached a number of conclusions regarding the algorithms and their respective performances. First and most evidently, the obtained results and previously studied theoretical properties indicate that the brute force algorithm and any of its derivates are not suitable for solving the knapsack problem, as it cannot handle instances with a large number of items. In most real-world applications of this problem such as finance or telecommunication traffic modeling, the number of "items" in the knapsack would result in astronomical times for finding an optimal solution, rendering this algorithm useless. However, for the purposes of this project, the brute force algorithm proved itself useful to find optimal solutions to generated instances of the problem and calculating accuracies.

Regarding the accuracies of the rest of the algorithms, we can conclude that there is a trade-off between the quality of the solutions and the algorithm's execution time. The best performing algorithms in this project were also the ones that took the longest to compute a solution. The only algorithm that appeared to not follow this conclusion was the fractional greedy algorithm, which appeared to have an overall very satisfactory accuracy of the solutions. Nonetheless, we believe it to be necessary to further experiment with this algorithm on much larger datasets, to assess the scalability of its performance on different magnitudes of data.

On the sensitivity of the algorithms to different aspects of the problem, such as the statistical distribution of the samples, we can conclude that the structure of the data can negatively impact the accuracy of the solution if the algorithm is limited in how it searches a solution, as it was observed in some of the instances of the dataset and the problem generator. This was the case for the greedy approaches, and even the genetic approach, as it was tested with a very limited number of individuals in every generation.

We can also conclude that while this study conducted an important number of experiments, it was quite limited in terms of parameters explored for each algorithm. As it was mentioned for the genetic algorithm, parameters such as the number of generations or the number of individuals could directly impact the accuracy of the solution found, and as such it merits further study. Other algorithms such as the FPTAS implementation can vary their accuracy depending on a chosen parameter, which needs to be tuned for any given problem.

Finally, we can conclude that this study successfully accomplished its objective of getting the members of this group acquainted with such a fundamental problem for computer science, and with a number of its possible solutions given a range of different algorithms studied in course.

References

[1] Hans Kellerer, Ulrich Pferschy, and David Pisinger. Multidimensional knapsack problems. In *Knapsack problems*, pages 235–283. Springer, 2004.

[2] Sami Khuri, Thomas Bäck, and Jörg Heitkötter. The zero/one multiple knapsack problem and genetic algorithms. In *Proceedings of the 1994 ACM symposium on Applied computing*, pages 188–193, 1994.

[3] Hazem AA Nomer, Khalid Abdulaziz Alnowibet, Ashraf Elsayed, and Ali Wagdy Mohamed. Neural knapsack: A neural network based solver for the knapsack problem. *IEEE Access*, 8:224200–224210, 2020.

[4] David Pisinger. Where are the hard knapsack problems? *Computers & Operations Research*, 32(9):2271–2284, 2005.

[5] Harvey M Salkin and Cornelis A De Kluyver. The knapsack problem: a survey. *Naval Research Logistics Quarterly*, 22(1):127–144, 1975.

Checklist Discussion

This section aims at commenting and discussing the achieved advances with regards to the punctuation grid given in the project prompt.

1. Did you proofread your report?
   - Yes, we proofread the report.
2. Did you present the global objective of your work?
   - Yes, the global objective of the work is presented in the first section of the report.
3. Did you present the principles of all the methods/algorithms used in your project?
   - Yes, we presented the principles of all the methods in Section 1.1.
4. Did you cite correctly the references to the methods/algorithms that are not from your own?
   - We provided a reference list of related works, but since we believe all the algorithms were made from scratch, we did not provide citations for individual algorithms.
5. Did you include all the details of your experimental setup to reproduce the experimental results, and explain the choice of the parameters taken?
   - Yes, we included all the details of the experimental procedure to reproduce each experiment. We attempted to provide an explanation for the choice of parameters in the section 2.4
6. Did you provide curves, numerical results and error bars when results are run multiple times?
   - Yes, we provided curves, numerical results and error bars when the experiments were run multiple times.
7. Did you comment and interpret the different results presented?
   - Yes, we commented on what we considered to be the most relevant results to the study.
8. Did you include all the data, code, installation and running instructions needed to reproduce the results?
   - Yes, attached to the report we included all the data, code, installation and running instructions to reproduce the results.

9. Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/techniques and debugging?
   - Yes, we attempted to unify both problem generation and data extraction to facilitate the addition of new methods and debugging existing ones.

10. Did you make sure that the results different experiments and programs are comparable?
    - Yes, we only compared experiments that we considered to have been executed in similar enough fashions to be comparable.

11. Did you sufficiently comment your code?
    - Yes, comments were provided throughout all the source code.

12. Did you add a thorough documentation on the code provided?
    - Documentation as such is not included in the code, as we believe the report and comments on the code is explicit enough.

13. Did you provide the additional planning and the final planning in the report and discuss organization aspects in your work?
    - We did not add planning-related aspects to the report as these would be presented in the defense.

14. Did you provide the workload percentage between the members of the group in the report?
    - We did not add planning-related aspects to the report as these would be presented in the defense.

15. Did you send the work in time?
    - With the exception of this list which was incorrectly sent with the report, it was sent in time.

Appendix

This section aims to briefly present instructions to execute the algorithms described in this report, and included as attachments to the compressed file delivered to the instructors. It also presents in the following pages, two previously presented tables that may have been difficult to read, given the size of the font.

How to use the attached files.

- In the same directory as this report, you will find a number of directories and python files. The directories correspond to the datasets used, and their corresponding optimal values or solutions depending on the dataset. The python files will be named according to their corresponding algorithm or function.
- To replicate any of the experiments mentioned on this report, the reviewer must have an operating version of Python 3.9 and any of the dependencies included at the beginning of each of these algorithms. Normally, these dependencies are limited to the NumPy library, the Pandas library, the Matplotlib library, and the Leven library.
- A working version of the files included in this compressed folder can be found on the Github repository "Arielogg/the_mldm_knapsack", but as we intend to keep working on these functions, changes may be performed after the date of delivery.

Instance / Property	**f1**	**f2**	**f3**	**f4**	**f5**	**f6**	**f7**	**f8**	**f9**	**f10**
n of samples	10	20	4	4	15	10	7	23	5	20
profit mean	41.2	54.25	12	10.25	37.1333	10.5	26.857142	839.52173	28.4	54.3
weight mean	53.9	54.9	6.75	4.75	48.86666	13	13.2857	844.6956	18.2	54.9
profit std	30.78	24.5903	2.236	2.680951	30.146015	7.04627	21.76122	212.2762	9.393614	24.590851
weight std	26.2885	29.7958	1.47901	1.92028	26.90196	10	9.61716	213.5113	7.52063	29.7958
profit skew	0.0873	-0.29548	0	-0.6568	0.511743	-0.16207	0.8283	-1.05138	-0.801976	-0.30157
weight skew	-0.3824	-0.27088	0.43465	-0.27803	-0.36168	0.1866	0.60016	-1.0855	0.473604	-0.27088
profit kurtosis	-1.434	-1.05508	-1.36	-1.098979	-0.9758	-1.609252	-0.47511	-0.852715	-0.894549	-1.052801
weight kurtosis	-0.74051	-1.26049	-1.15428	-1.4266	-0.730976	-1.32235	-0.94377	-0.816026	-0.6377	-1.26049
capacity	269	878	20	11	375	60	50	10000	80	879
optimal	295	1024	35	23	481.0694	52	107	9767	130	1025

Table 8: Main statistical properties of the low-dimensional dataset.

Instance	**Accuracy of Algorithms (%)**							
	BaB	**Randomized**	**Genetic**	**V-Greedy**	**W-Greedy**	**F-Greedy**	**Dynamic**	**Polynomial**
f1	85.08474	99.66101	99.322	97.6271	72.5423	100	100	100
f2	81.34765	100	84.375	100	92.6757	100	100	100
f3	94.2857	80	100	80	100	100	100	100
f4	100	95.6521	100	100	69.56521	69.565217	100	95.6521
f5	72.7545	99.15409	71.7152	99.154	71.5073	100	100	100
f6	96.1538	100	96.15384	82.6923	96.15384	100	100	100
f7	89.7196	95.3271	86.91588	100	73.8317	95.327102	100	100
f8	99.6723	99.92833	99.35497	99.9795	97.8191	99.8566	100	99.89
f9	100	71.53846	100	100	100	100	100	100
f10	91.4146	100	89.26829	100	92.68292	99.41463	100	100

Table 9: Accuracy results of certain approaches when testing on the low-dimensional dataset.