Advanced Algorithms: Knapsack Problem

Computation with multiple approaches

Group's members :
Christiane Manuela AYO NDAZO'O
Landry BAILLY
Chadapohn CHAOSRIKUL
Gloria ISEDU
Felipe CORTES JARAMILLO

December 2022

1 Introduction

The knapsack problem has been known since the end of the century 19[1], thanks to George Ballard Mathews. This problem has many applications, first in an asymmetric encryption algorithm introduced by Martin Hellman, Ralph Merkle and Whitfield Diffie in 1976. It is also used for investment management support systems, to optimize the loading of boats or planes and other applications. But this problem is best known as an optimization problem to fill a bag, which justifies its name.

This problem has different variants like the 0/1 knapsack problem, the multiple knapsack problem, the unbounded knapsack problem, etc.

The knapsack problem is considered an NP-hard problem. This means that we can find a polynomial reduction of a problem in NP[1](in this case it is the sum of subsets problem) to the knapsack problem. Moreover this problem is NP, which means that it can be solved with polynomial complexity, given a non-deterministic Turing machine. This means that we can check whether a solution is correct or not in polynomial time. Indeed, for example for the problem of the backpack 0/1 [5], it suffices to sum the weight of all the chosen objects and to compare it to the maximum weight (so the complexity is O(n)). But we do not know if it is possible to have a polynomial algorithm in time to find this solution, it depends on the famous question, is P equal to NP?
To summarize, the knapsack problem is NP and NP-hard, so it is an NP-complete problem [4].

[1]A problem belongs to class NP if there is a polynomial checker for this problem

Saying that the knapsack problem is NP-complete does not mean that it is impossible to solve it. But we know that solving it using combinatorial approach is almost impossible because of the exponential time complexity. Fortunately, a heuristic approach also makes it possible to solve this problem in a polynomial time. So, that is the main goal of this project. **Solve the knapsack problem using a heuristic approach** with algorithms like :

- Dynamic programming
- Greedy approaches
- Branch and bound algorithms
- Randomized algorithms
- Ant colony
- Genetic programming

A combinatorial approach has also been implemented with a **brute-force algorithm**. Besides the different implementations, each one of them has an analysis, in order to compare which one of them is best to a given scenario. Therefore, we will talk about the 0/1 and the 0/1 multiple knapsack problems. We will first present the problem in mathematical terms. Next, we will present all the algorithms that we have made. Finally, we will compare them and get some meaningful insights of each approach.

2 Folder Structure

In the root folder we can see that there is a folder called "TestingModule", useful for experimentation, we will explain it later. The others correspond to the variants of the knapsack problem that we studied.

Then, for each knapsack problem, there are 5 different folders :

- Algorithms : contains all the algorithms implemented to solve the problem.
- classes : contains a python class with all the attributes and methods useful to represent the problem. It's basically the representation of the knapsack problem for the computer.
- Generator : The generator that create randomly input files as instances.
- Input : contains data files, the benchmarks and the generated ones. Each file correspond to a knapsack instance.
- Output : contains the results returned by the algorithms in comma separated values (csv) files.

All the code, libraries requirements, installation guide are contained in the GitHub repository: `https://github.com/Manuela-AYO/AlgoProject2022`.

3 0/1 Knapsack Problem

3.1 Explanation of the Problem

The 0/1 Knapsack Problem is a variant of the knapsack problem defined as :

$$\begin{aligned} &\text{maximize} \sum_{i=1}^{n} v_i x_i \\ &\text{subject to} \sum_{i=1}^{n} w_i x_i <= KW \text{ and } x_i \in \{0,1\}. \\ &\text{where } n = \text{number of items} \\ &v_i = \text{value of item i} \\ &w_i = \text{weight of item i} \\ &KW = \text{Capacity of the knapsack} \end{aligned}$$

These notations are going to be used in all the 0/1 knapsack problem section.

We have a set of items which have values and weights. We want to put those items in a sack which has a capacity KW. For each item i pushed in the sack, $x_i = 1$ and for the others $x_i = 0$. The objective with the knapsack problem is then to maximize the values in the knapsack with respect to its capacity. [5]. In the following sections each algorithm will be described in detail.

3.2 Theoretical Analysis

3.2.1 Brute Force Algorithm

This algorithm is quite simple. The purpose is to test all the possible answers.

Number of possible answers Each answer corresponds to a certain combination of objects. Suppose we want to take k objects, with $k <= n$, and n the total number of objects. So we have to test all the combination of k among n, whatever the order. The mathematical formulation of this is:

$$C_n^k = \frac{n!}{(n-k)!k!}$$

Because we want to test all the possible answers, we also have to do this whatever k in $[\![0; n]\!]$. (Theoretically speaking, the answer with k=0 can be useful if all the others give a weight superior to the maximum allowed.)

So, the total number of possible answers is

$$\sum_{k=0}^{n} C_n^k = \sum_{k=1}^{n} \frac{n!}{(n-k)!k!}$$

Thanks to the Newton binomial theorem, which is

$$\sum_{k=0}^{n} C_n^k x^k y^{n-k} = (x+y)^n$$

We can simplify the precedent formula (with x = y = 1)

$$\sum_{k=0}^{n} C_n^k = 2^n$$

Algorithm principle For each possible answer, we calculate the weight and the value. In the end, we return the best answer, which means the answer with the greatest value, given the fact that its weight is less than the maximum allowed.

Complexity The algorithm is composed of a loop that will test all the possible answers. For each of the 2^n iterations, it calculates the answer's value, the answer's weight, and the next combination. All of these functions are $O(n)$. Therefore, the total time complexity of the algorithm is $O(n * 2^n)$.

3.2.2 Dynamic Programming

The dynamic programming approach divides problem into smaller sub-problems. Then, sub-solutions are derived, and stored in memory to be used repeatedly later.

3.2.3 Bottom-Up Dynamic Programming

Algorithm Principle The bottom-up version starts solving the problem from the beginning. It executes from item 1 to item n, where n is the total number of items in the knapsack. Meanwhile, the top-down version starts from the end, and executes from item n to item 1.

The bottom-up version as in [1] has been implemented.

Dividing the problem into smaller sub-problems means that substructure of the problem and the recursive principle are needed to be defined. To do this, we followed [3]:

- Define $OPTBU(i, w)$ as the maximum value that is retrieved from the bottom-up algorithm, from the subset of items 1 to i, with the sum of weights equal to w. There are two possibilities:
  - The item i is not included in the optimal solution:
    The $OPTBU$ takes the best value from subset of items $\{1, 2, .., i-1\}$ with the sum of weights w as:

$$OPTBU(i, w) = OPTBU(i-1, w). \quad (1)$$

– The item i is included in the optimal solution:
The $OPTBU$ takes the best value from subset of items $\{1, 2, .., i-1\}$ with the new sum of weights $w - w_i$ as:

$$OPTBU(i, w) = v_i + OPTBU(i-1, w-w_i) \tag{2}$$

Given v_i as the new item value.

The new sum of weights is equal to w - w_i because we reduced our knapsack capacity with the item weight w_i.

- Define the recursive principle :

$$OPTBU(i, w) = \begin{cases} 0 & \text{if } i = 0 \\ 0 & \text{if } w = 0 \\ OPTBU(i-1, w) & \text{if } w_i > w \\ max(OPTBU(i-1, w), v_i + OPTBU(i-1, w-w_i)) & \text{if } w_i <= w \end{cases}$$

Given the item i, the sum of weights w, and the item weight w_i.

To compute $OPTBU(i, w)$ by the recursive principle, a table for storing the sub-solutions is constructed with the sum of weights as columns and the items as rows. And as the base conditions of the recursive principle take 0 if $i = 0$ or $w = 0$, the first row and the first column are filled with 0 as:

$$OPTBU(0, w) = 0 \tag{3}$$

In fact, $i = 0$ means that we don't select any item and $w = 0$ means the sack capacity is 0.

Hence, the columns range are $\{0, 1, ..., KW\}$ and the rows range $\{0, w_i, ..., w_n\}$. Moreover, the size of the table is equal to $(n+1)$ rows multiply by $(KW+1)$ columns.

Complexity The complexity of the algorithm is $O((n+1)(KW+1))$, where n is the number of items in the knapsack, and KW is the knapsack capacity. Recall that $n+1$ and $KW+1$ come from the fact that the first row and the first column of the table are filled with 0 as base conditions. The space of the algorithm is also $O((n+1)(KW+1))$.

After all of the cells are filled, the items taken by the algorithm are traced based on [1]. The tracing algorithm starts the bottom-right most cell $OPTBU(n+1, KW+1)$ in a recursive manner, as following:

- The item is taken if the value of the current cell is greater than the cell above.
- If the item is taken, traverse to row $i-1$ and column $w - w_i$. Compare this cell with the cell above, decide whether to take this item or not.

- If the item is not taken, traverse to row $i-1$ and column w. Compare this cell with the cell above, decide whether to take this item or not.
- Keep going until the number of remaining item is equal to 0

Recall that w is the current sum of weights, w_i is the current weight, and i is the current item.

The complexity of the tracing function is equal to $\mathrm{O}(n+1)$, where n is the number of items.

As the quality of solution is measured by given the time, the algorithm waits until the last cell of the current row is computed, and the position of the last cell is returned. The idea is to avoid tracing uncomputed rows, which are filled with 0.

3.2.4 Top-Down Dynamic Programming

Algorithm Principle Define $OPTTD(i, w)$ as the maximum value retrieved from the subset of items 1 to i with the sum of weights equal to w. To compare between the two versions, the top-down algorithm firstly computes the top-most state $OPTTD(n, KW)$. Meanwhile, the bottom-up algorithm firstly computes the bottom state $OPTBU(0, w)$,

While solving the top-states, if the same sub-problems are called, the algorithm reuse sub-solutions, which are stored in the table, such that to avoid recomputing. This technique known as memorization [6].

The optimal substructure and the recursive principle of the top-down version are the same as in the bottom-up version.

The top-down version from [7] has been implemented. In our version, the first row of the top-down table is filled with 0. So that, when item i_1 is equal to 0 and the item i_0 is equal to -1, we can avoid taking item i_1.

Complexity The complexity of the algorithm is $\mathrm{O}((n+1)(KW+1))$, where n is the number of items in the knapsack, and KW is the knapsack capacity. The space of the algorithm is $\mathrm{O}((n+1)(KW+1))$, in addition of the recursion stack: $\mathrm{O}(n+1)$.

The top-down version uses the same tracing function as the bottom up version.

3.2.5 Greedy Algorithm

Greedy algorithms aim at finding the best solution by taking items that will either maximize the profit gained by the selection or taking items with the least weight to maximize the number of things that go in the knapsack.

3.2.6 Greedy by Value

Here, the greedy algorithm wants to maximize the profit gained by the knapsack, so it selects the highest value and if the corresponding weight of the value is smaller than the knapsack size, then the item is selected.

Complexity Since the most significant time complexity in this algorithm is the loop needed to select and item then check if the total selected items' weights are still less than KW, we conclude that the complexity is O(n) and the algorithm runs in linear time.

3.2.7 Greedy by Weight

Here, the greedy algorithm wants to maximize the number of items that go into the knapsack, so it selects the lowest weights in a iterative way, while the current sum of weights is less than the size of the knapsack.

Complexity Since the most significant time complexity in this algorithm is, once again, the loop needed to select an item, then check if the total selected items weights are still less than KW, we conclude that the complexity is O(n) and the algorithm runs in linear time.

3.2.8 Greedy by Ratio

Here, the greedy algorithm wants to maximize the ratio of the profit to be gained to the weight of the item, so it gets the ratio of value to weight for all the items, then selects the item with the highest ratio while the knapsack weight is not exceeded.

Complexity Since the most significant time complexity in this algorithm is the loop needed to select and item then check if the total selected items' weights are still less than KW, we conclude that the complexity is O(n) and the algorithm runs in linear time.

3.2.9 Creation of a new greedy algorithm : sort by ratio and converge

This algorithm is a creation using the principle of greedy by ratio. First let's talk about the main idea of the algorithm, then we will explain the last improvement of this idea. The python file 01Knapsack\Algorithms \Greedyratio_sort_and_converge.py correspond to the classical algorithm and the python file 01Knapsack\Algorithms \Greedyratio_sort_and_converge_2.py correspond to the improved algorithm.

Algorithm principle In this algorithm, we first sort the objects with their respective ratio Value / Weight. Then, we take all the first ones in order to fill the weight condition as for the greedy by ratio algorithm. That process creates an initial solution, let's call it $(a1_j)_{1\leq j\leq n1}$ with n1 the number of objects in this answer. Then it calculates the ratio V / W of this answer, which is

$$\sum_{j=1}^{n1} Value(a1_j)/MaximumWeight$$

We divide by the maximum weight of the problem instead of the sum of the weight of all the objects of the answer to penalise answers that have unused capacity. Of course, $\sum_{j=1}^{n1} Weight(a1_j) < MaximumWeight$
We plug this new ratio in our dataset with all the objects. Then we sort again by ratio, but this time, for all the objects that have been chosen previously we take into account their new ratio.
We continue in this way until convergence. Which means, we stop when the new answer give a Value less than or equal to the precedent one.

Example Let's take an example with a tiny knapsack problem. There is only 5 data and the maximum weight is 10.

Values	Weight	V/W
19	2	9,5
45	9	5
5	1	5
4	1	4
30	8	3,75

Figure 1: Data at the first step

After the first step of the algorithm, we choose the first row, we don't select the second one because its weight is too heavy ($2 + 9 = 10$). Then we choose the third one and the fourth one. The last one is also too heavy. The value of our answer is $19 + 5 + 4 = 28$. So, the new ratio is $28/10 = 2,8$. For the next step we sort again all the objects using the new value.

Values	Weight	V/W	New ratio
45	9	5	
30	8	3,75	
19	2	9,5	2,8
5	1	5	2,8
4	1	4	2,8

Figure 2: Data at the second step

For this new answer, we chose the first object and then fourth one, the two others are two heavy. So we have a new answer with a better value equal to 50. The third step won't change this result, so this is the final result provided by the algorithm.

Complexity At each step, the algorithm complexity is

$$T(Step_i) = T(sortvalue) + T(selectobject) + T(calculatenewratio) + constant$$

The complexity of the sorting algorithm we used (pandas.Data Frame.sortvalue()) is $O(n^2)$ in the worst case. The complexity to select the current answer and to calculate the new ratio is $O(n)$. And the rest of the operation have a $O(1)$ complexity.
So $T(Step_i) = O(n^2)$
Theoretical speaking we have locked the number of step to $2 + n^2$. But, in reality we never had more than 5 steps, whatever the value of n.
So the theoretical complexity of this algorithm is $O(n^4)$. However, in practice it seems to be $O(n \cdot log \cdot n)$. More of this in the analysis section.
Additionally, the pre-process complexity is $O(n)$ because we need to calculate all the ratio V / W for each object.

New Improvement This new improvement increases the quality of the algorithm but it also increases its complexity.
It runs the previous algorithm once. It will give an answer. It takes the first object of this answer, which has the best ratio V / W given all the chosen objects. This object is selected for the final answer.
Then, we create a knapsack sub-problem without this object and with a new maximum weight equals to the last, minus the weight of the selected object.
It runs again the previous algorithm with this new knapsack sub-problem and repeat the process.
We stop it when the final answer is reached, with the supposition that we cannot add a new item without exceeding the limit. So there will be at most n big steps, if the final answer can accept all the items. The new complexity is $O(n^2 \cdot log \cdot n)$

3.2.10 Fully polynomial time approximation scheme(FPTAS)

Algorithm Principle The assumption of FPTAS is that finding an optimal solution is costly because the values are too high. Then, we round the values by a factor δ defined by :

$$\delta = \epsilon * \frac{V}{n}$$

$$\begin{aligned} \text{where } V &= highest\ value\ of\ the\ items \\ \epsilon &= error \in [0, 1] \\ n &= total\ number\ of\ items \end{aligned}$$

There, we have a parameter ϵ which is the error that we allow for the solution provided by the algorithm. The solution should then be at least $(1 - \epsilon)\cdot$optimal_solution.

After the rounding, FPTAS finds the best value using the dynamic programming algorithm.

Complexity FPTAS is polynomial in the number of items but also in ϵ. That's the reason why its complexity is $O(nV/\epsilon)$.

3.2.11 Branch and bound algorithm

Algorithm Principle The Branch and Bound (BB) algorithm aims to find the optimal solution through the most promising paths. These paths are built according to a tree structure.

For the knapsack problem, we first turned the maximization problem into a minimization problem, simply by multiplying by *-1*. Next, we built a binary tree whose nodes are items and the 02 branches representing 0 if an item is not selected and 1 else. Then BB will prune the tree according to the most promising branches.

To achieve this objective, 02 important aspects must be taken into consideration : the **cost** and the **upper bound**.

1. **The upper bound**
   It represents the minimum value [2] that we can get from a path in the tree. This value is computed at the beginning as follows :

   $$u = \sum_{i=1}^{n} v_i * i$$

   s.t

   $$\sum_{i=1}^{n} w_i * i <= KW$$

   And during the branch and bound process, we recompute the upper bound on a path only when we investigate the case where an item is not selected. To compute the upper bound on those nodes, we consider all the values that we got before studying the item i and we add up the upper bound, now moving from the item at position i+1 to n(i+1 being the position of the next object). Finally, the upper bound is updated if the upper bound of this path is less than the actual upper bound.

2. **The cost**
   The cost is what is lost on a path. It's this parameter that BB uses for pruning and also to determine priority paths. The cost has pretty much the same formula, but one thing differs.

[2] it is in reality the maximum value. Remember that we have turned the maximization to a minimization problem

Suppose that due to the weight remaining on the bag, there can only be 2 out of 5 items that could go in. In addition, the sum of the weights of these items is strictly less than the remaining weight. To calculate the cost, we will add the values of these items but also, taking into account the following:

$$cost = \left\{ \begin{array}{l} W'/W_i * p_i \\ with\ W' : remaining\ weight, \\ W_i : weight\ of\ the\ next\ item\ in\ the\ list\ of\ items \\ p_i : value\ of\ the\ next\ item\ in\ the\ list\ of\ items \end{array} \right\}$$

Having the cost, we don't explore a path if its cost is greater than the upper bound of the tree.

Also, this version of branch and bound is the **Least cost branch and bound(LC-branch and bound)**, where we first explore a path for which the cost is the least.

Complexity Finally, the value is reduced to $O(n^2)$.

3.2.12 Randomized Algorithm

Algorithm Principle For the randomized algorithm we designed a local search schema as mentioned in [9], including a random factor to select the next step in each iteration of the algorithm. The whole procedure begins with the initialization of a random solution, which is a random selection of objects that are going to be included in the knapsack. This selection is represented as a random list of zeros or ones. At first, this solution can be viable, which means that all the weights inside the knapsack are lower than the maximum weight condition. However, it can also be possible that the algorithm presents an initial selection which exceeds the maximum weight. Therefore, is not valid as a solution for the knapsack problem. Once we generate the initial selection, we measure the quality computing the sum of all the values and weights inside the knapsack. Then, we check whether the condition is exceeded, followed by two possible rules:

$$Randomized(CurrentKnapsack) = \begin{cases} Add, & \text{if } weights < condition \\ Remove, & \text{otherwise} \end{cases}$$

Based on the previous condition, if all the weight's of the current knapsack do not exceed the maximum weight condition, we can add a new element. Otherwise, it means that the current knapsack selection exceeds the solution. Therefore, it is not valid and we need to remove one element to lower the total sum of the weights.

When it is feasible to add a new element, we first get the list of which objects are not included in the current knapsack. In other words, the objects that are

represented in the list with a zero. Then, for each one of them, we calculate their value and we sort them in descending order, keeping the most valuable elements at the beginning of the list. Based on the concept of elite selection in the genetic algorithms [2], we want to pick the best possible candidates in order to improve the solution. So, in this scenario, we pick among the most valuables elements in order to maximize the value of the knapsack. This candidates represents a small group which size is conditioned by a parameter called selection ratio. Moreover, this group is extracted from the first elements of the list. To put it differently, from the possible elements which will increase the current value of the solution the most. To give an example, if there are ten possible elements to add and the best possible candidate group has a length of 3, we are going to pick one randomly among the first three elements of all the the possible candidates.

In the other hand, when we want to remove an element, the procedure is similar as the mentioned above. However, the selection will be based on the elements that are currently included in the knapsack, the ones that are represented in the list with a one. Also, the sorting in this case will be ascending, in order to pick randomly a element that will not reduce a lot the current value. The selection is similar as the addition function, choosing randomly from a small group of the best possible candidates.

Complexity Finally, at each iteration of the algorithm we will check whether to add or remove an object, until we run out of time or iterations. This algorithm has a total time complexity (in O notation) of $O(n \cdot log(n))$, where n is the total number of objects in the knapsack. Moreover, this algorithm in some scenario can get really good answers as well as really bad ones, depending of the initial random initialization and how many iterations (or time) is provided, this will be explained in depth when seeing the experiments results.

3.2.13 Ant Colony Algorithm

Algorithm Principle The idea behind the ant colony algorithm relies in the construction of a solution with the contribution of multiple threats of execution. Where, each one of this threats will be references as an ant. This construction is done in collaboration using information that relies global for all the ants, which are the pheromone trail τ and the attractiveness of the move μ. The first one, keeps track of the objects that were selected in the partial solution made by the ants, rewarding the objects that were selected and punishing the ones that were not. Meanwhile, the second is the value that represents each object when picked, this can vary depending on the implementation of the algorithm as shown in [8]. Finally, the iterative construction will create a neighbourhood of possible solutions. In other words, a solution of objects that must be part of the solution and others that will not be viable. So, for each new iteration, there will be a base selection that each ant will work with.

To begin, the algorithm makes an initialization of the pheromone trait and the attractiveness of the move. The first one is set equally to all the objects in order to keep the probability of picking any object similar, assigning the highest

value to each object. In the other hand, the attractiveness of the move μ is initialized as:

$$\mu_j = \frac{v_j}{w_j}$$

Which boils down to the ratio between the value and weight of each element of the knapsack. At this point, we start one iteration of execution of the algorithm where each ant based on the current pheromone trait and attractiveness of the move, constructs a solution picking different elements of the knapsack. The selection of each ant is based on a probability p_j, which is the following:

$$p_j = \begin{cases} \frac{\tau_j \mu_j}{\sum_{j \in N} \tau_j \mu_j}, & \text{for j} \in \text{N} \\ 0, & \text{for j} \notin \text{N} \end{cases}$$

Where τ is the pheromone trait, μ is the attractiveness of the move, j is an element of the knapsack and N is the current ant partial solution.This probability can be explained as follows: for a potential element that the ant will pick, it is the average of the products between the pheromone trait and attractiveness of the potential selected object over all the products of the objects that are currently in the ant knapsack solution. This probability will be done for each object until there are no more options left. Among all the answers constructed by the ants, the best of them will be set as the best solution for the iteration. Furthermore, with this solution we will check if it is the best answer so far and, we will update the pheromone traits based on the following condition:

$$\Delta\tau = \frac{1}{1 + \frac{N_{best} - N}{N_{best}}}$$

Which means that those objects that were included in the best iteration solution, are kept with a higher probability than the ones that were not selected. This will impact the next iterations for the decisions that each ant will make. Also, to improve the convergence of the method, it was introduced a variable called decay, which will multiply the objects that were not selected, reducing even more the probability to be picked in the next iterations.

Finally, the algorithm will stop if one of these conditions is met. First, if the algorithm reaches the number of iterations or the time of executions, it will stop and return the best solution so far among all the iterations. Moreover, if the solution of a current iteration is close to the current best solution, it means that the algorithm is reaching a local optima. In other words, it converged to a local optima solution. This convergence rate is measured as follows:

$$C_{rate} = \frac{N_{best} - N}{N_{best}} <= Tolerance$$

If this ratio C_{rate} is lower than the tolerance, it means that the solution is good enough to be returned. Therefore, we stop the algorithm. Otherwise, it means that we can still iterate until a better solution is found.

Complexity The algorithm has a time complexity (O notation) of O($m \cdot n^2$), where m is the number of ants and n the amount of objects in the initial knapsack.

3.2.14 Genetic Programming

Algorithm Principle The idea behind genetic programming algorithms is to use the concept of survival of the fittest to get an optimal solution.[10] @online

Much like in survival of the fittest, we have an initial population made up of *individuals* which are randomly generated possible 0/1 knapsack solutions. An individual consists of genes and chromosomes. A chromosome is a string that contains information about the individual (i.e the sequence that the solution consists of) while a gene is a character inside the chromosome.

That is,

$$gene = \{0, 1\}$$

and the fittest among them will crossover to produce more fit offspring. Then, we mutate the offspring by a random factor to add a bit of variation to the population and the offspring become the new population and the process repeats itself.

In adapting the genetic programming algorithm to fit the 0/1 knapsack problem, we do the following:

1. Initialize the population: we initialize the population with individuals whose chromosomes are generated randomly.

2. Calculate the fitness of an individual: we check the fitness of each individual by calculating the chromosome weight and checking if the chromosome weight is less than the weight of the knapsack. The chromosome weight is calculated by multiplying each gene in the chromosome by its corresponding weight and getting the sum of these operations(Recall $gene = \{0, 1\}$).

   Formally,

   $$cw = \sum_{i=1}^{n} g_i * w_i$$

   where,

   $$\begin{aligned} cw &= \text{chromosome weight}, \\ n &= \text{length of chromosome}, \\ g_i &= gene, \\ w_i &= weight \end{aligned}$$

If the chromosome weight is more than the size of the knapsack (which means that for this possible solution, the weights selected are more than the size of the knapsack) then we set the fitness of this individual/possible solution to zero. Else, we store the chromosome's value as the fitness.

$$if \sum_{i=1}^{n} g_i * w_i \leq kw,$$
$$fitness = \sum_{i=0}^{n} g_i * v_i;$$
$$else$$
$$fitness = 0;$$

where,

$$n = \text{length of chromosome},$$
$$g_i = gene,$$
$$v_i = value,$$
$$w_i = weight,$$
$$KW = \text{knapsack weight}$$

3. Select parents: Using the fitness scores gotten from the previous operation, we select parents as the individuals with the highest fitness scores

4. Crossover: We then cross the fit parents, gotten from the previous stage, to produce more fit offspring. the crossover is done for a pair of parents at a time, so for every two parents, the chromosome will be divided into two equal parts and be crossed to produce two offspring such that each offspring will inherit equal chromosomes from the two parents as shown below.

Parent 1	1	1	0	0	1	1	0	0	0	0
Parent 2	1	0	0	0	1	0	1	1	1	1

Offspring 1	1	1	0	0	1	0	1	1	1	1
Offspring 2	1	0	0	0	1	1	0	0	0	0

5. Mutation: we mutate the offspring by a random factor so that there is variation in the population. We set a mutation rate between 0 and 1 and generate a random number between 0 and 1. If the number generated is greater than the mutation rate, nothing happens, else, we chose a random index and invert the value of a gene in the chromosome

Complexity The time complexities for the operations carried out are: initializing the population takes O(1) time, calculating the fitness takes O(n) time, selecting parents takes O(n) time, the crossing over takes O(n) time and the mutation takes O(n) time. However, recall that all the listed operations will also happen for each new generation, creating a nested loop, so the total time complexity is $O(n^2)$ or more literally, $O(n \cdot 4n)$

3.3 Experimentation

In order to test the different algorithms, it was necessary to have different criteria to evaluate and compare performance. First, some benchmarks that have been a basis of truth for our algorithms. These benchmarks were extracted from `http://artemisa.unicauca.edu.co/~johnyortega/instances_01_KP/`, which consists of 31 knapsack instances: 10 small instances and 21 large-scale instances. Beyond the established issues, we developed an instance builder. Thus, we can have specific instance sizes to test algorithms, this program will be explained shortly.

In order to fairly compare the algorithms, two metrics were taken into account: first, the **quality of the solution**. In other words, which algorithm provides the best value within a given running time. On the other hand, the **running time**. This metric consists of how much time each algorithm takes to provide the best possible value.

Based on these two principles, a test module was created to automate the testing process of all algorithms and facilitate comparison between all available instances.

To be able to reproduce the test, all the commands executed to assess the algorithms are also provided in the source code.

Also, in order to avoid external factors, all the experiments below have been done using the same environment, which consisted in a Intel core i5 CPU with 3.10GHz of frequency, and 10 GB of RAM. Moreover, all executions were done independently.

3.3.1 Instances Generator

In order to test our algorithms, we implemented a **generator**.

A generator is a python module which aims to create different instances of the knapsack problem; an instance being a particular configuration describing the knapsack problem.

Four parameters were important to provide those instances :

- the number of items : it is basically the total number of items to study for the knapsack problem
- the sack weight : it is the maximum capacity of the sack
- the range : it represents the maximum weight that an item can have. This value is in the interval [1, knapsack weight - knapsack weight/4].

The upper bound of this interval has been inspired from known instances of the knapsack problem

- the distribution : it is the type of correlation between values and weights of the items. We have 3 kinds of distributions :
  - **Uncorrelated instances(un)** : the values are weights come from different uniform distributions
  - **Weakly correlated instances(wc)** : values are function of weights plus a noise
  - **Strongly correlated instances(sc)** : there is a bijection between values and weights

3.3.2 Testing Module

Principle In order to test all our algorithms, we created a testing module. This testing module allow the user to test any algorithm. It can be executed through a shell command. Also, It will call the each algorithm, run it with a specific instance and give the answer in a comma separated file (csv).

How to use it In the folder \TestingModule, run the file 01Knapsack.py :
python ./01Knapsack.py "some parameters"
It is the testing module for 0/1 Knapsack Problem. About the parameters, if you write -h it will explain each one of them. Let's take a look at each one of them:

1. The name of the output csv file inside the folder \01Knapsack\Output. The answer of the test will be written here.
2. The name of the algorithm that we want to test. It can be : BruteForce, BranchAndBound, RatioSortGreedy, ValueSortGreedy, Weight-SortGreedy, RatiosortAndConvergeGreed, BottomUpDynamicProgramming, Randomized, FullyPolyNomial, GeneticProgramming or AntColony
3. The type of file where the instance is contained, "t" for text file (column split with ' ') or "c" for csv file (column split with ',')
4. The name of the input instance file inside the folder \01Knapsack\Input. This folder must contain our knapsack problem, meaning the number of item, the maximum weight and all the items.
5. The maximum theoretical value. If you don't know it, write "-".
6. The maximum time allowed. If there is no limit, write "-".
7. The maximum iteration allowed. If there is no limit, write "-".
8. (optional) –sp1 the first specific parameter, useful for some algo
9. (optional) –sp2 the second specific parameter, useful for some algo
10. (optional) –sp3 the third specific parameter, useful for some algo

Answer Format The answer will be add to the output csv, it take only one line. Let's see the meaning of each line :

1. The name of the tested algorithm.
2. The name of the input instance file inside the folder \01Knapsack\Input.
3. The number of item inside this instance.
4. The maximum weight condition of this instance.
5. The maximum theoretical value or "-".
6. The maximum time allowed or "-".
7. The maximum iteration allowed or "-".
8. The executed time.
9. The value of the answer. ($\leq$ Maximum theoretical value)
10. The weight of the answer. ($\leq$ Maximum weight condition)
11. The number of item of the answer
12. All the specific parameters or "-".

Create bash file In order to do large tests, several bash and shells files were created in order to contain plenty executions of the test module. In the repository are contained some examples of the files created.

3.3.3 Dynamic Programming

To benchmark the algorithms, the running time and the efficiency are assessed.

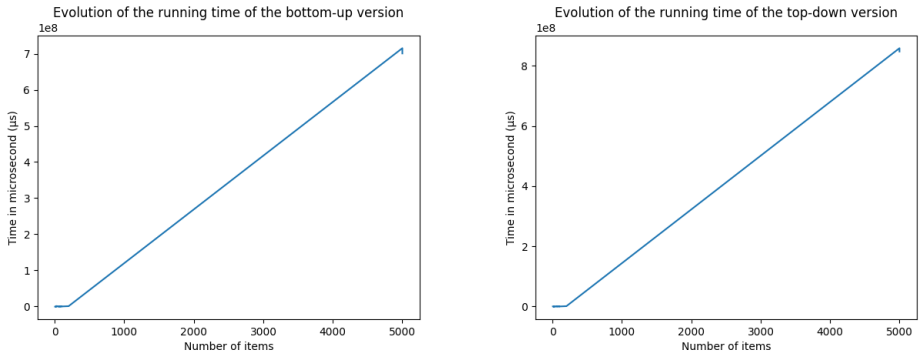

Figure 3: Evolution of the running time of dynamic programming in microsecond over the number of items.

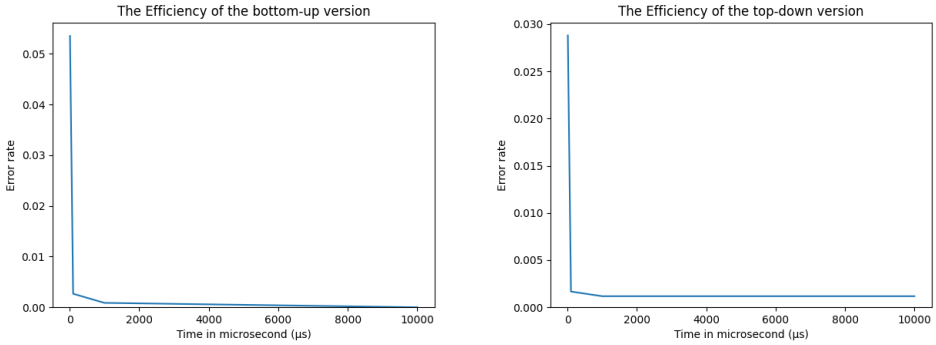

Figure 4: The efficiency of dynamic programming in microsecond over time

1. Running time assessment

   Analysis

   Overall, the running time of both versions increases linearly as shown in the **Figure 3**, left for the bottom-up version, right for the top-down version. In dynamic programming, when the number of items and the knapsack capacity increase, the size of the table also increases. Hence, it takes more time for the algorithm to run. This follows by the complexity of both versions, which is O((n+1)(KW+1)), where n is the number of items and KW is the knapsack capacity. Moreover, the **Figure 3** also shows that the top-down version is slightly slower than the bottom-up version. The reason is that the top-down version takes extra time for the recursive stack.

2. Error rate assessment

   Analysis

   In **Figure 4**, the efficiency over time in microseconds of dynamic programming on artificial data is shown. Note that the error rate comes from the average of the difference between the optimal value and the average algorithm value, normalized by the optimum value at the given time.

   The plot on the left shows the efficiency of the bottom-up version. It states that the error decreases as time passes. After few microseconds, the error decreases sharply to 0. This shows that if there is enough time, dynamic programming gives the optimal solution.

   The plot on the right shows the efficiency of the top-down version. The error also decreases as time pass. However, the error does not reach to 0 due to the fact that 10,000 microseconds is not enough for large instances.

3. Some important decisions made

   - In the top-down version, if the redundant sub-solutions got called, the values of sub-solutions are reused, to reduce the steps of calculation.

4. Improvements

   - We realized that dynamic programming can be done in a parallel manner to improve the running time. For example, the memorization table can be computed simultaneously on distributed systems.

5. Conclusion

   - **Advantages:** Dynamic programming gives an optimal solution. Moreover, it runs fast on small ranges of items. However, on the large dimensional instance such as 5,000 items, it did pretty well as it took only around 11 minutes to find a solution.
   - **Disadvantages:** When the number of items increases, the size of memorisation table also becomes greater. This leads to memory crash, especially when the number of items is equal or greater than 10,000.

3.3.4 Greedy Algorithm by Value

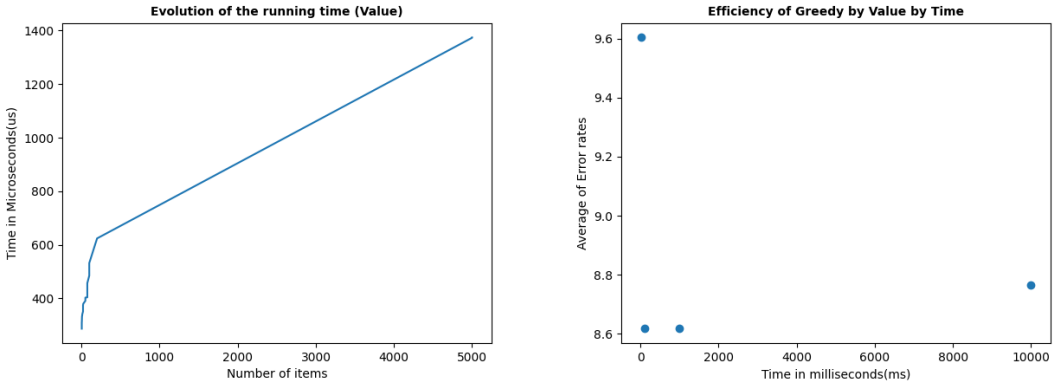

Figure 5: Running time(left) and error rate(right) of the Greedy algorithm by value

1. Running time assessment
   The plot at the left of **Figure 5** evaluates the evolution of the running time(in microseconds) of greedy algorithm by value with respect to the number of items. We deduce that the running time starts out logarithmic when faced with a small number of items but starts to increase linearly as the number of items increase dramatically.

2. Error rate / Efficiency
The plot at the right of **Figure 5** evaluates the evolution of the error rate with respect to the time in milliseconds. The error rate is the difference between the optimal value and the value found by the algorithm at a given time. Because the greedy considers how high the value is before taking the weight, if the values and weights are correlated, then the error rate will be low and else the error rate will be high, as a result, since the plot is an average of error rates, we can't see if the algorithm's error rate decreases or increases over time.

3. Some important decisions made
Through different tests, we got relevant information which made us make some important decisions like :
• Use the highest value directly, instead of first sorting the values, to add an item to the knapsack if knapsack weight is not exceeded. This approach takes away the extra complexity of having to sort the items first and it produces the exact same result with this step removed.

4. Improvements
An improvement over this specific algorithm will be to perform the greedy algorithm by ratio of $values : weights$ which is also done.

5. Conclusion
• Advantage : we see that this algorithm is very fast as it computes 5000 number of items in 1400 microseconds which is 0.0014 seconds, and it is also a very straight forward algorithm to implement.
• Disadvantage : it does not give the optimal solution.

3.3.5 Greedy Algorithm by Weight

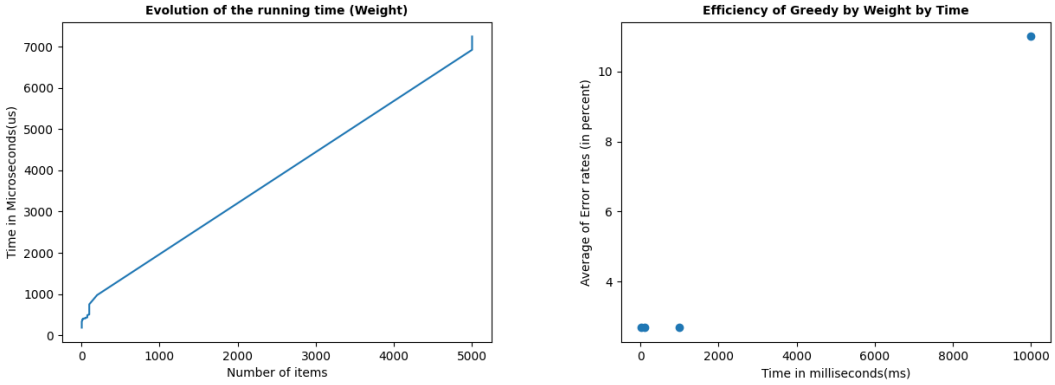

Figure 6: Running time(left) and error rate(right) of the Greedy algorithm by weight

1. Running time assessment
   The plot at the left of **Figure 6** evaluates the evolution of the running time(in microseconds) of greedy algorithm by weight with respect to the number of items. We deduce that the running time linearly increases with the number of items.

2. Error rate / Efficiency
   The plot at the right of **Figure 6** evaluates the evolution of the error rate with respect to the time in milliseconds. The error rate is the difference between the optimal value and the value found by the algorithm at a given time. Because the greedy considers how low the weight is before taking it, if the values and weights are correlated, then the error rate will be low, else the error rate will be high. However, if the algorithm is collecting as many weights as it possibly can, it is also collecting a lot of values, which is why the error rate is low and it performs similarly when given more time, but recall how the error rate is calculated, meaning that the value given by the optimal algorithm at 10000 milliseconds is so much better that the one given for this algorithm at the same time.

3. Some important decisions made
   Through different tests, we got relevant information which made us make some important decisions like :
   • Use the lowest weight directly, instead of first sorting the weights, in order to add an item to the knapsack if the knapsack weight is not exceeded. This approach takes away the extra complexity of having to sort the items first and it produces the exact same result with this step removed.

4. Improvements An improvement over this specific algorithm will be, much like in the greedy by value, to perform the greedy algorithm by ratio of $values : weights$ but again, as previously mentioned, this is also done.

5. Conclusion
   • Advantage : we see that greedy by weight is fast, computing 5000 items in about 7000 microseconds or 0.007 seconds, but it is slower than the greedy by value algorithm. Also, the error rate, while constant, is considerably lower than the greedy by value algorithm.
   • Disadvantage : The error rate is constant at an average value, which means that even when given more time, it cannot produce the optimal solution.

3.3.6 Greedy Algorithm by Ratio

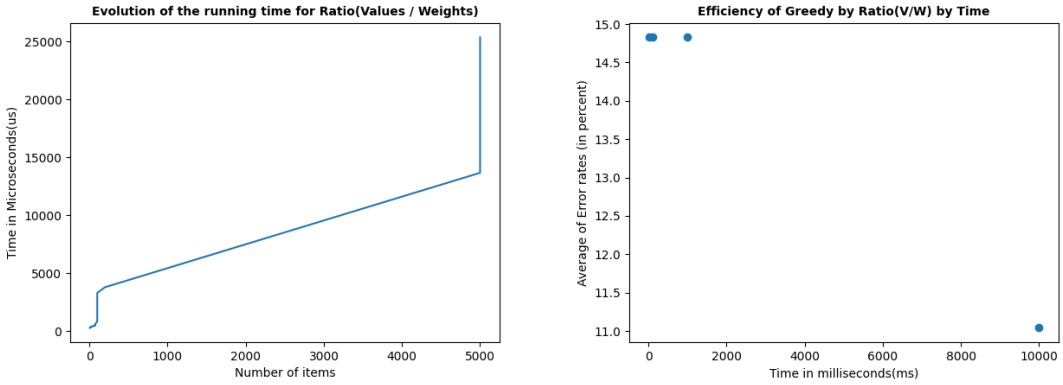

Figure 7: Running time(left) and error rate(right) of the Greedy algorithm by ratio

1. Running time assessment
   The plot at the left of **Figure 7** evaluates the evolution of the running time(in microseconds) of greedy algorithm by value with respect to the ratio of the values of items to their corresponding weights. We deduce that the running time grows linearly when computing close to 5000 items but once the number of items becomes 5000, the time it takes to compute increases dramatically in a non linear way.

2. Error rate / Efficiency
   The plot at the right of **Figure 7** evaluates the evolution of the error rate with respect to the time in milliseconds. The error rate is the difference between the optimal value and the value found by the algorithm at a given time and the plot shows that the error rate of the greedy algorithm by ratio remains constant initially, then decreases when given more time, irrespective of the time given to run. Which once again means, it's optimal value will not change even if it is given more time to compute.

   Because of how the error rate is calculated, the efficiency of the solution given by this algorithm is determined by what value the algorithm with the optimal solution gave.

3. Some important decisions made
   Through different tests, we got relevant information which made us make some important decisions like :
   • Use the highest ratio directly, instead of first sorting the ratios, to add an item to the knapsack if knapsack weight is not exceeded. This approach takes away the extra complexity of having to sort the items first and it produces the exact same result with this step removed.

4. Improvements An improvement over this specific algorithm will be to perform a modified version which involves converging as will be discussed in the next section.

5. Conclusion
   • Advantage : we see that greedy by ratio is also takes less than a second to compute 5000 items, while actually being slower than its predecessors. the best solution but is also fast on small ranges of items.
   • Disadvantage : it is slower in greater ranges of items

3.3.7 Greedy sort by ratio and converge

1. Running time assessment

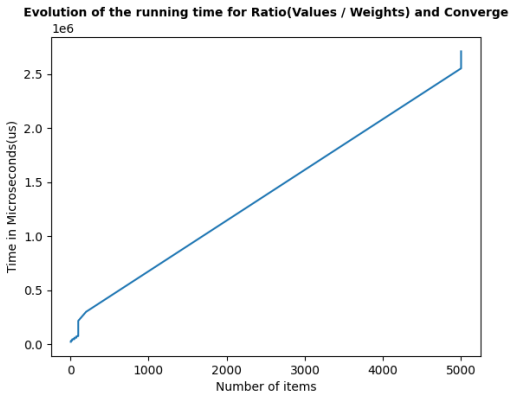

Figure 8: Running time - Improved Greedy Algorithm

This algorithm is very fast, for all the test, it take at most 3 seconds, for less than 5000 data as we can see in the previous graph. The time unit is $10^6 milliseconds = seconds$

2. Error rate / Efficiency

The quality of the algorithm does not depend of the number of elements in the knapsack. In general case it is quite well with less than 0.5 % error. For the one with 18 % we can explain it because it is a low dimensional data set (instances with few objects), and missing one good object for the answer really penalise the error rate. (figure 9).

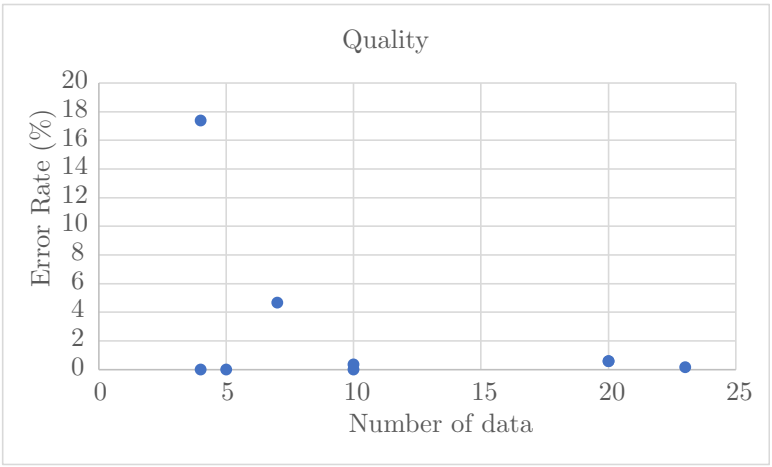

Figure 9: Quality of sort by ratio and converge

3. Some important decisions made This algorithm was improved by selecting for each process the best value and reset the algorithm with a knapsack sub problem (see on previous sections).

4. Improvements

   The second graph, in **Figure 10** shows the relative error (%) and the first one absolute error. In fact, the quality of the result do not depend of the number of data, and the theoretical value. It depend of the number of items in the theoretical result that the algorithm missed. Whatever is the representation, this graphs shows that the improvement corrected the most important error. However, there is one drawback. With 5000 elements, the algorithm took 3 minutes and 6 minutes to find the answer instead of 3 second. So, one way to avoid this gaps in time, the algorithm was simplified. In fact, deleting all the elements with a tiny ratio V/W. Let's take the example in **Figure 11**.

   As shown in **Figure 11**, the end of the first step gave this result. So, the first value was kept for the other step. Moreover, It was deleted all the object with a lower ratio than the lowest chosen object. In this case, the last object will be delete from the data set. Then for the next step, the data set will have a smaller size and the algorithm will run faster. (Because sometimes, it delete 0 objects. However, the theoretical complexity of the algorithm do not change).

   After this second improvement, the algorithm take 1 minute and 40 seconds running, which is much faster. But the improvement did not improved the overall algorithm, because of the bad results. As you can see in **Figure 12**, the quality didn't change for the last file and became worse for the first one. Apparently some items with little ratio V/W were chosen

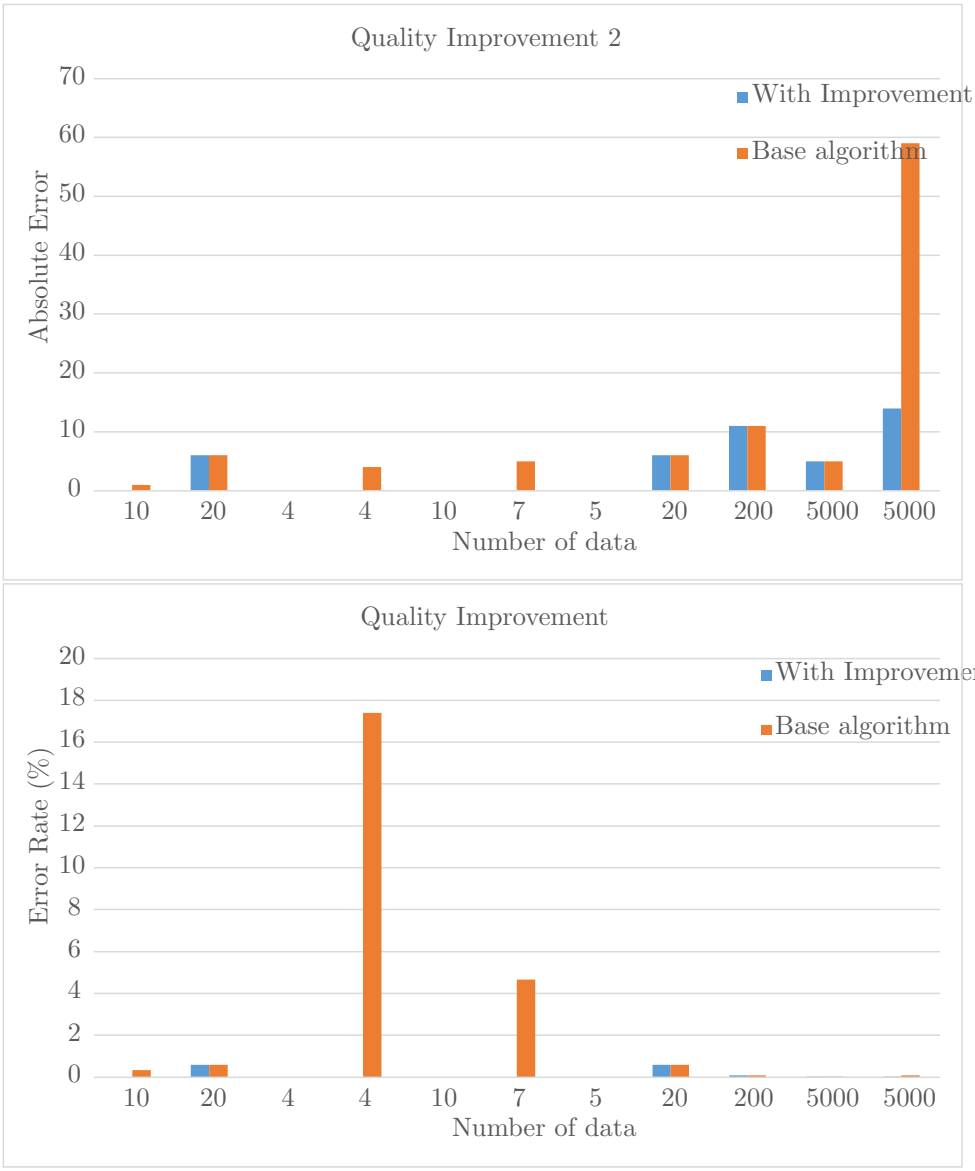

Figure 10: Quality comparison

in the theoretical optimal result. I finally decided to abandon this way and keep the precedent algorithm, giving the result of **Figure 12**.

5. Conclusion This new algorithm is quite efficient and very fast for the smaller data sets, despite the time complexity that can bring the new implementation, it generates better estimates.

Values	Weight	V/W	New ratio
45	9	5	
30	8	3,75	
19	2	9,5	2,8
5	1	5	2,8
4	1	4	2,8

Figure 11: Data at the second step

Figure 12: Quality comparison

3.3.8 Branch and bound algorithm

As said before, branch and bound algorithm is supposed to give the optimal solution. Thus, we decided to evaluate the running time of the algorithm and its error rate.

1. Running time assessment

Analysis

Figure 13 evaluates the evolution of the execution time (in microseconds) of the branch and bound algorithm with respect to the number of items.

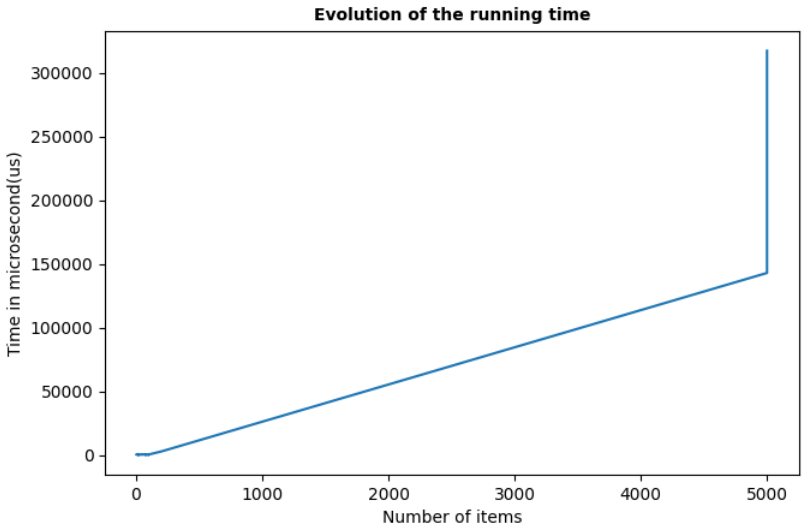

Figure 13: Running time of branch and bound algorithm

We deduce that the execution time increases linearly with the number of items with a kind of elbow point at 5000 items.

This increase can be explained by the size of the exploration tree which becomes longer when we increase the items. Remember that in branch and bound, nodes of the tree are decisions on the objects with 0 if an item is not selected and 1 else.

2. Error Rate/Efficiency

Analysis
Figure 14 evaluates the evolution of the error rate with respect to the time in milliseconds(ms).

We see that the error rate of branch and bound is pretty constant and equal to 0.

This result could be explained by the fact that branch and bound always perform the same operation : evaluate only the node for which we don't take an item.

Also, branch and bound seems to be pretty fast because it always performs the operations within the given time.

3. Some important decisions made Through different tests, we got relevant information which made us take some important decisions like :

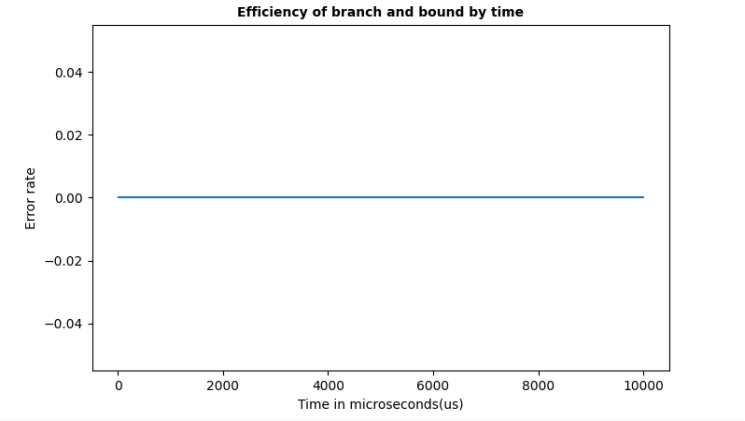

Figure 14: Efficiency of branch and bound algorithm

- **Use the least-cost branch and bound(LC-branch and bound)** instead of the **LIFO** or **FIFO** branch and bound. This approach prioritizes the cost of a node over the order in which the tree is traversed; which prunes faster the tree
- **Sort the table by decreasing ratio value/weight** : At the beginning, due to the efficiency of LC-branch and bound, we thought it was not relevant to firstly sort the table before diving into the branch and bound algorithm. But with tests on small instances of the knapsack, we noticed that the pruning of the tree was longer, means there where much nodes to proceed before finding the best solution. That's the reason why before branching, we sort the items.

4. Improvements
   By this time, we(the group) didn't find a better way to approach the knapsack problem.

   Improvements here are more questions that we asked ourselves than anything else.

   - The evaluation of a node : we saw that the evaluation of the node is polynomial in time. So, **is it possible to get a cheaper algorithm to approximate upper bounds and cost in order to better deal with large number of items ?**
   - Sort of the array by decreasing ratio value/weight : the question is mainly **Is it possible to minimize the number of nodes/solution to study without firstly sort the table ?**

5. Conclusion

- **Advantage :** we clearly see that branch and bound, not only find the best solution but is also fast on small and medium ranges of items.
- **Disadvantage :** it is slower in greater ranges of items

3.3.9 Fully polynomial time approximation scheme(FPTAS)

We evaluated the running time and the efficiency of FPTAS algorithm.

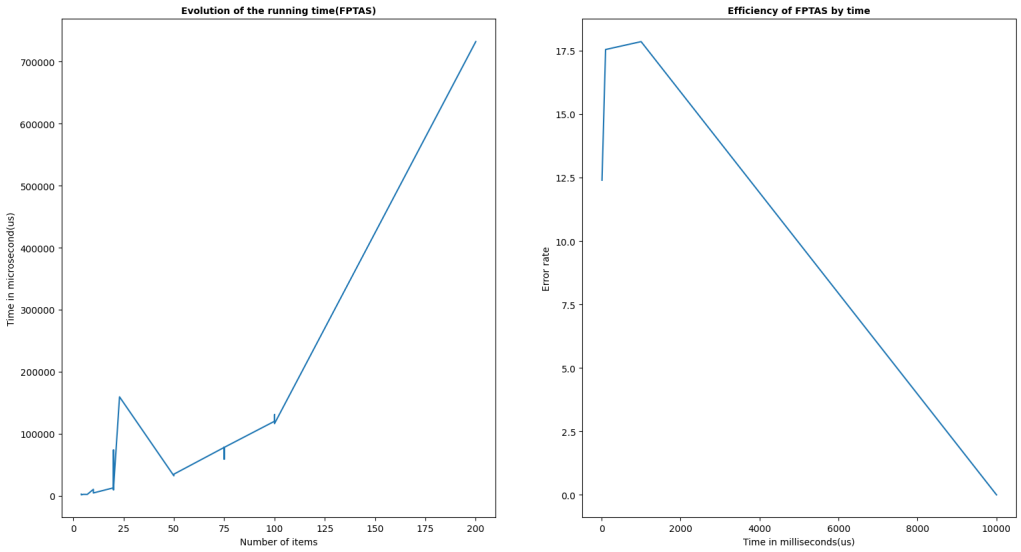

Figure 15: Running time(left) and error rate(right) of FPTAS algorithm

1. Running time assessment
   Analysis
   The plot to the left of **Figure 15** evaluates the evolution of the running time(in microseconds) of FPTAS algorithm with respect to the number of items. We observe on this plot a pic after 25 items then it decreases to finally grow linearly.

   Those variations are due to the type of instances of the knapsack that we have. With this plot, we deduce that FPTAS is sensitive to the distribution behind the knapsack problem.

   Our tests show us that for instances of the knapsack where ranges of values were much smaller than the capacity of the sack, FPTAS takes more time because the parameter *delta* won't be enough large to round down the values.

2. Error Rate/Efficiency
   Analysis

The plot to the right of **Figure 15** evaluates the evolution of the error rate with respect to the time in milliseconds.

At the beginning we observe a fluctuation due to the fact explained on the evaluation of the running time.

But after that, the error rate decreases very quickly over time and get closer to 0. This shows that scaling doesn't have a huge effect on the efficiency of the algorithm. Remember in fact that FPTAS is a rounded version of dynamic programming which always return the best solution. So, depending of the factor ϵ, FPTAS can return solutions which are not that far from the optimal ones.

3. Some important decisions made
   Through different tests, we got relevant information which made us take some important decisions like :

   - **The choice of** ϵ : we tested different values of ϵ and we observed that with $\epsilon = 0.001$, we have a pretty good result, which in addition is not too consuming for hard-drive resources.
   - **Use of the recursion** : As FPTAS is based on dynamic programming, it's more intuitive to use loops. But we decided to use recursitivity because it only computes cells of the array that can contain parts of the best solution

4. Improvements
   We realized that some improvement could be done on this algorithm :

   - Use a tree : by using tree, we could parallelize the computation which will save time.

5. Conclusion

   - **Advantage :** rounding down the values improve the running time
   - **Disadvantage :** this scale may be useless on some instances of the knapsack

3.3.10 Randomized Algorithm

1. Running time assessment
   Regarding the time assessment with the randomized algorithm There are some important insights according to the following table:

   We can see that when we consider the column of time (which is in minutes), the longer the algorithm executes, the better the answer gets. There are some special cases when the algorithm already starts with a good solution. However, in most of the scenarios we can see the slow increase in the average value regarding the time taken to execute.

Number of elements in the knapsack	Time Executed (min)	Theorical Best Optimal Value	Algorithm Max Value
100	0:10:00	254	211
	1:40:00		208
	16:40:00		209
75	0:10:00	504	337
	1:40:00		382
	16:40:00		414
50	0:10:00	121	112
	1:40:00		108
	16:40:00		112
20	0:10:00	236	202
	1:40:00		208
	16:40:00		221

Table 1: Randomized Algorithm - Time Assessment

As seen in the theory of the algorithm, each iteration manages a small change in the current solution, which helps improving the current solution to a better estimate, but its convergence to the optimal value expected is rather slowly. Moreover, the impact of one iteration affects one element in the knapsack, so we would expect many iterations to at least modify the initial solution.

Regarding the time and the value relationships, the algorithm will take at least O(n) time in order to extract a solution which does not exceed the weight condition. Because with this time, the initial solution should at least be checked in one pass, so it can fix the elements that exceeds the weight. Moreover, as we increase the iterations from O(n) we are going to get a better estimate. Because we will do plenty changes trying different new elements without exceeding the max weight condition. Therefore, the complexity will increase to O(n*log(n)) when we want a good estimate from the randomized algorithm.

2. Analysis

Number of elements in the Knapsack	**Optimal Value**	**Algorithm Max Value**
200	4082	11238
23	7975	9767
20	534	1025
10	124	295
7	101	107
5	96	130
4	26	35

Table 2: Randomized Algorithm - Values among knapsack sizes

It is important to mention that the algorithm was executed with a maximum value of iterations of 200. Therefore, as we increase the number of Knapsack elements, the value brought by the algorithm will be worst. However, when we have several elements lower than the number of iterations performed by the algorithm, we will get a good estimate close to the optimal value. This behavior can be explained with the fact that the

algorithm in each iteration performs one action: add element or subtract element. Therefore, when we execute the algorithm with a small number of iterations, the estimate will not be good.

In the large-scale knapsack (above 200 elements) the random algorithm requires a lot of iterations to converge to a solution. At least to expect a valid solution the number of iterations should be the number of elements in the knapsack (O(n) time). Moreover, if we execute the algorithm with more iteration compared to the number of elements, we will start improving the solution in most of the cases, as explained in the time assessment.

3. Some important decisions made
   Here are some important decisions made in the algorithm:

   - **The choice of the parameter condition rate** : We tested different values of the parameter and we got that the best value overall was 0.3. Because if the number is high, we will end up picking randomly elements from all the knapsack and if it is too small, the decision will always be biased by a few elements.
   - **Single action per iteration** : One important factor is that at each iteration we can perform only one action, which can generate a slow convergence to the optimal value.

4. Improvement
   We realized that some improvements could be done on this algorithm :

   - Multiple actions per iteration: One idea is to flip different elements at the same time or adding/removing multiple elements, in order to have more efficient iterations.
   - Escape local optima: One solution if the algorithm value does not change meaningfully, we will change the initial solution with a new one. So, we can start doing a local search from a different starting point.

3.3.11 Ant Colony Algorithm

1. Running time assessment
   Regarding the time assessment with the ant colony algorithm. There are some important insights comparing the time with the number of elements in the knapsack.

   The time taken in the small instances gets almost linear, meaning that with few iterations the algorithm is getting a good estimate of solution. However, when there are more elements in the knapsack, the algorithm increases its time in an exponential form, which can be explained by the theoretical complexity behind the algorithm which is O(mn^2), where m is the number of ants and n the amount of objects in the initial knapsack.

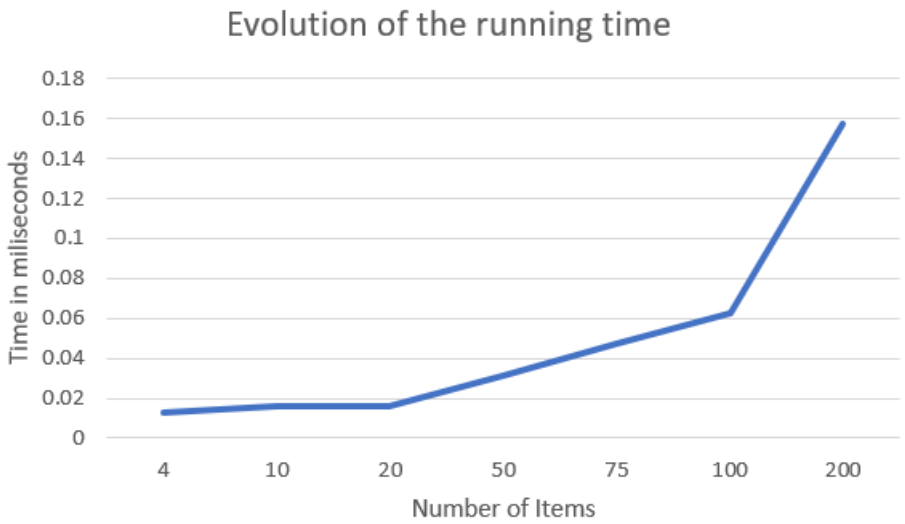

Figure 16: Evolution of the running time(milliseconds) with the number of items

Number of elements in the knapsack	Theoretical Optimal Value	Algorithm Average Value
5000	72505	29185
	44356	26402
200	11238	4273
23	9767	7885
10	295	262
	52	46
7	107	102
5	130	100
4	35	31

Table 3: Ant Colony Algorithm - Value among elements

2. Analysis

In table 3 we can see the average values brought by the algorithm, this tests were made in the same conditions with 100 ants, 200 iterations, and decay rate of 0.5. We can see that as we increase the size of the knapsack, the algorithm starts to return an average solution which is lower than the optimal value expected. Once insight is that the convergence of the algorithm depends in multiple factors, including the pheromone trail, how it is updated, the amount of influence in the decision taken and the random factor of taking a solution or not.Therefore, there can be done several modifications in the pheromone trail and increasing the number of iterations that will increase the average value brought by the algorithm. However, this will bring a slow time complexity because there will be more actions performed in total.

3. Some important decisions made
   Here are some important decisions made in the algorithm:

   - **The choice of the parameters in the algorithm** : Among the main parameter to be tuned there are the number of ants and the decay rate. Through several test we measured that for low dimensional instances that have a size between 1 to 200 elements, the best value for the ants was 75. Furthermore, beyond 200 elements the value will increase slowly. Taking the maximum quantity of elements in the benchmark, the greater size was 5000, for this 100 did a better performance than the other values for the parameter. In the other hand, with the decay rate, this value presented a best performance when tuned at 0.5. Which in other words, the probability to pick an elements that was not selected in the previous iteration is reduced by a half. A greater or lower value for this parameter would not generate better solutions.
   - **Addition of a tolerance measure** : This algorithm relies strongly in the number of iterations. However, there is the question of when to stop? When we measure the quality of the solution with a tolerance, we can get a solution that will have quality enough to be good and the time will not be great. With this, the tolerance used for all the test was 0.01.

4. Improvement
   Some improvements that were extracted from the algorithm performance:

   - The initialization of the attractiveness of the move μ can be done in multiple ways: with the ratio of the weight and value, just the value or other heuristics. So, changing the initialization criteria may have an impact in the final result of the algorithm.
   - Some references of the algorithm uses two parameters called alpha α and beta β, which are used to measure if the ant decision will rely more in the pheromone trail or the attractiveness of the move. In this algorithm both values have the value of 1, which means that both elements should be considered in the same proportion. However, adding this parameter and tuning it will change the output of the algorithm.
   - In order to make faster the convergence, the decay rate can be removed and the pheromone traits will decay until certain number of iterations. Which is going to make some solutions more feasible and other less likely to be picked.
   - This Algorithm can also be paralleled. Where each ant will be a threat of execution. For this, it is necessary to implement a traffic light stop in the pheromones trait. In order to avoid the modification of a variable shared across multiple threats.

3.3.12 Genetic Programming

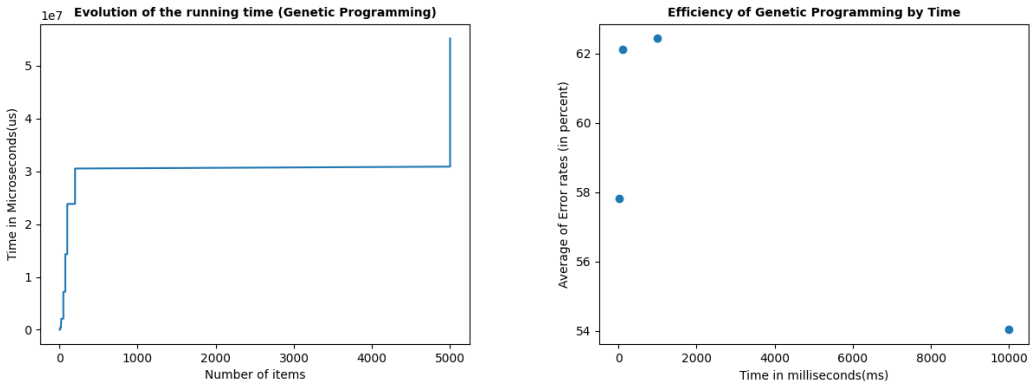

Figure 17: Running time(left) and error rate(right) of the Genetic programming Algorithm

1. Running time assessment The plot at the left of **Figure 17** evaluates the evolution of the running time(in microseconds) of the genetic programming algorithm with respect to the number of items. We deduce that the running time grows exponentially as the number of items in the knapsack grows.

2. Error rate / Efficiency The plot at the right of **Figure 17** evaluates the evolution of the error rate with respect to the time in milliseconds. The error rate is the difference between the optimal value and the value found by the algorithm at a given time. The plot shows that over time, the error rate will decrease but in fact, the algorithm behaves like random guessing such that it might improve or get worse in the next iteration so even in the particular instance it was run if we had plotted another point for time larger than 10000 milliseconds, the error rate could increase or decrease.

3. Some important decisions made Through different tests, we got relevant information which made us make some important decisions like :
   • The algorithm behaves like a coin flip so with different values the number of generations to have, the final solution will be good or bad with equal chances of it being good or bad, therefore, the number of generations was set to 50 but the results are just as random as when the number of generations is set to 10 or even 2.

4. Improvements
   By this time, we(the group) didn't find a better way to improve the genetic approach for the knapsack problem.

3.4 Overall Comparison

In the following section all the algorithms will be compared based in two metrics, the time and lastly, the value returned.

3.4.1 Time assessment

The main idea behind the time measure is letting all the algorithm run until they found the best solution possible. Furthermore, see which ones can execute in the lowest time possible compared to the other algorithms. First, the instances extracted from the data set were executed with the condition to have small quantity of elements, the results were the following:

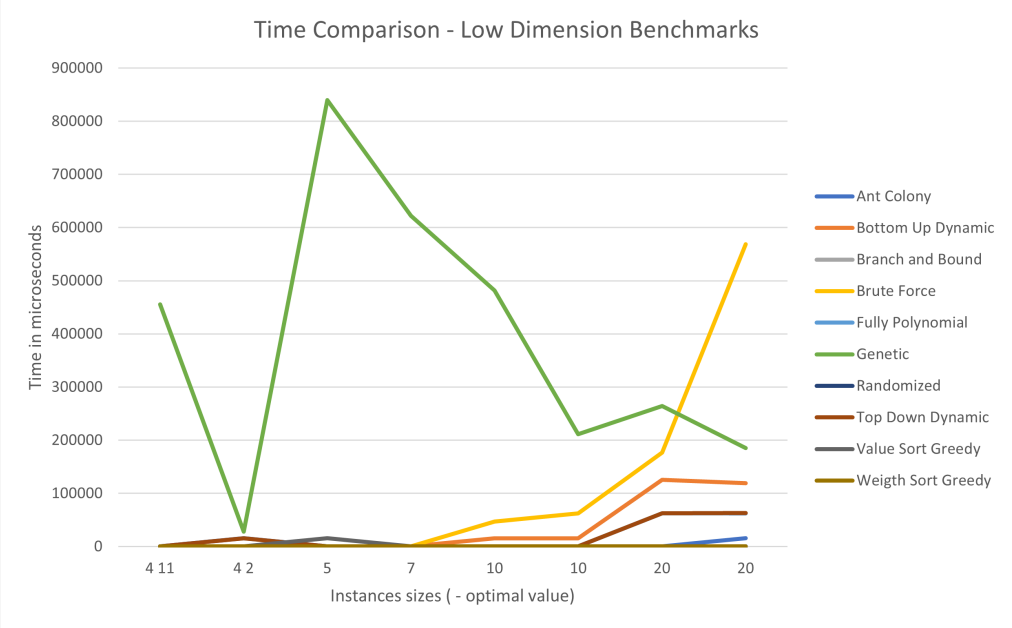

Figure 18: Time comparison - Low Dimension Benchmarks

A note from the figure above is that all algorithms that use number of iterations for their execution were standardized to 200. Among all the algorithms, the brute force, genetic and dynamic programming require the most time for their execution. Moreover, the genetic approach exceeds the brute force because 200 iterations cause the algorithm to keep mutating the population. However, as we increase the instances we can see that the time of the brute force algorithm starts to increase faster than any other algorithm. In this type of instances, the time is not a condition to get a good solution. Therefore, the algorithms that can bring a good estimate for the solution should be the best candidates to solve the instances.

For the next assessment, we took the instances of high dimensions which their size were greater than 100 elements. In this test the brute force, dynamic and fully polynomial time algorithms were removed because their time was high compared to the other algorithms. As we want to see which algorithm can execute in the lowest time possible, the three algorithms were totally removed.

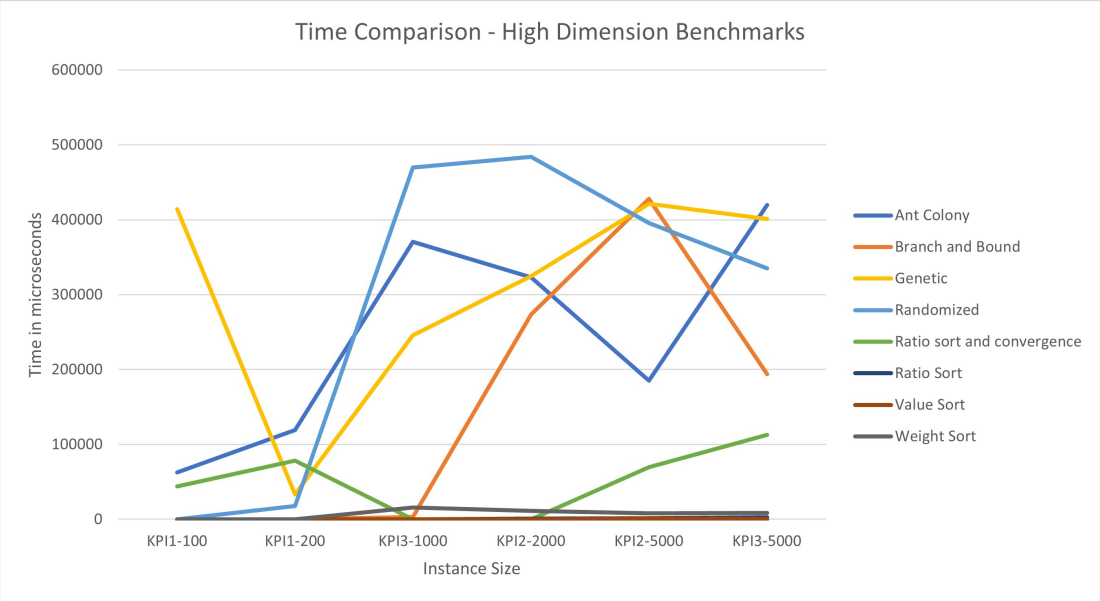

Figure 19: Time comparison - High Dimension Benchmarks

In high dimensions the algorithms that used iterations were executed with 500 iterations. Also, as we expect by theory, all the greedy approaches were able to get an answer with the lowest time possible but some other algorithms as the ant colony, genetic and branch algorithms were able to get a solution in a time lower than 1 second. Regarding high dimension instances the greedy approaches in term of times will be a good way to obtain a solution with a good estimate in a reasonable time, compared to algorithm that can bring the optimal value but will take a long time to execute.

Finally, instances created with the generator were tested in **Figure 20**, including examples from 20 elements to 100 elements. Furthermore, it was included the different distributions : uncorrelated instances(un), weakly correlated instances(wc), strongly correlated instances(sc).

We can see how the greedy approaches and branch and bound remain almost with a constant time among the executions compared to algorithms that take longer procedures as the fully polynomial or the dynamic approach. An important insight is that the correlation plays an important role in the time taken, if the elements of an instance are strongly correlated the time will be higher compared to uncorrelated elements, in term of values. As mentioned in

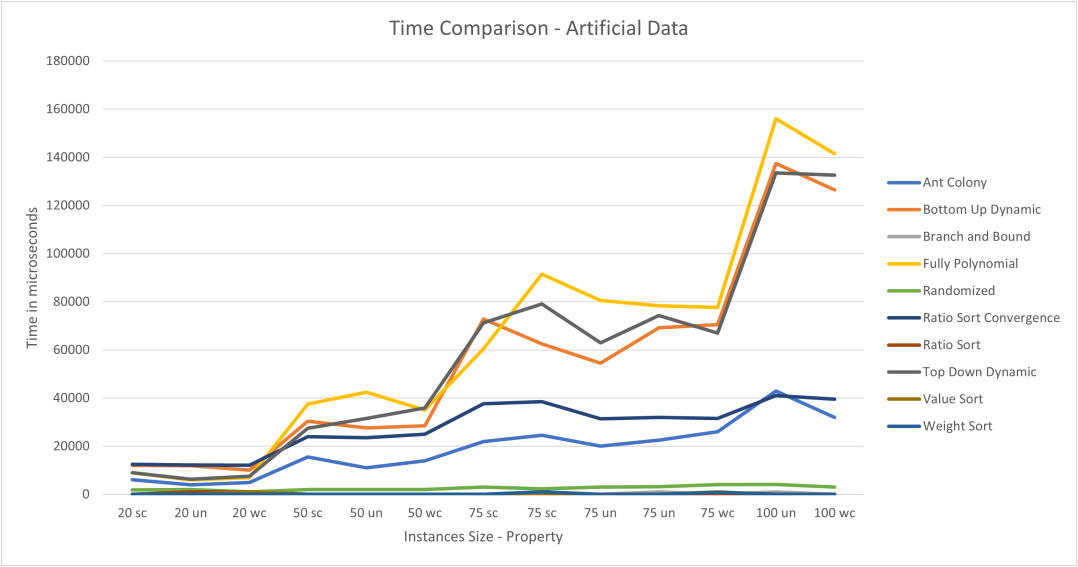

Figure 20: Time comparison - Artificial Instances

the previous scenario, when facing low dimensions instances the best pick in term of time will be a greedy algorithm. However, the solution will not be as good as other algorithms, so in small to medium instances an algorithm like dynamic programming, branch and bound or ant colony will be the most suitable candidates. Based on this insights in the running time, let's see the performance regarding the quality of the solution.

3.4.2 Quality of solution assessment

In this section, the quality of solutions provided by the different algorithms are measured with respect to the instances. To be able to assess that quality, we decided to base our study on the error rate given by the formula :
$errorRate = \frac{|output - optimalvalue|}{optimalvalue}$

In **Figure 21** and **Figure 22**, we see that greedy by ratio and converge, branch and bound, and fully polynomial time algorithm have the lowest error rate, which is pretty constant. Those results are explained by the structure used by those algorithms as explained above.

On the other side, randomized algorithm and ant colony have the highest error rates depending on the type instances. It's not a gradual increasing but mostly a unstable evolution. This behavior is explained by the randomness of those algorithms which doesn't allow the control of their parameters.

Now let us focus on the impact of the distribution on the quality of solutions via the artificial data.

At the first sight, the error rate seems to be constant apart of genetic algo-

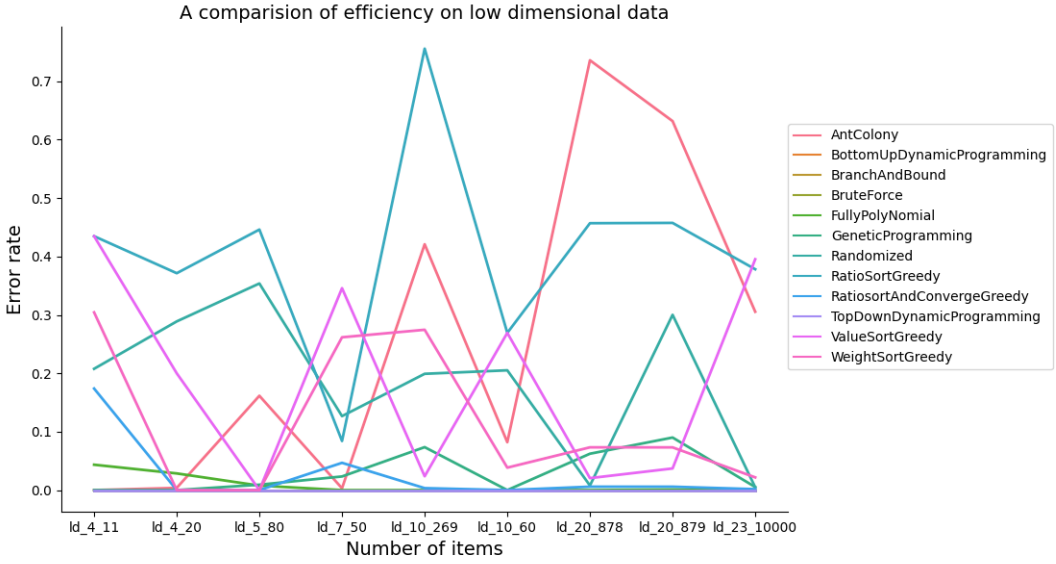

Figure 21: A comparison of efficiency on artificial data among all algorithms

rithm whose error rate clearly explodes compare to the other algorithms.

But a better view shows us all the algorithms are sensitive to uncorrelated data. In fact, the **Figure 21** and **Figure 22** on their error rate plots come from the instances which were uncorrelated.

In conclusion, the uncorrelated distribution of weights and values on an instance has a negative impact on the quality of the solutions.

4 Multiple Knapsack Problem

The multiple knapsack is another variant of the knapsack problem, the definition is similar to the 0/1 single knapsack. However, we have j sacks of capacities c1, c2, c3... cm. And, n objects to select, which each one has a profit v_i. Finally, the weights of the objects distributed in the sacks should all respect the weight conditions of each sack individually, and we search the distribution of elements that yield the maximum profit [5]. For this derivation we implemented few algorithms: brute force, greedy, dynamic and randomized algorithms.

4.1 Theoretical Analysis

4.1.1 Brute Force Algorithm

Principle Basically, it is the same principle as 0/1 knapsack problem. But this time, for each possible answer, meaning for each sub-part of the data-set,

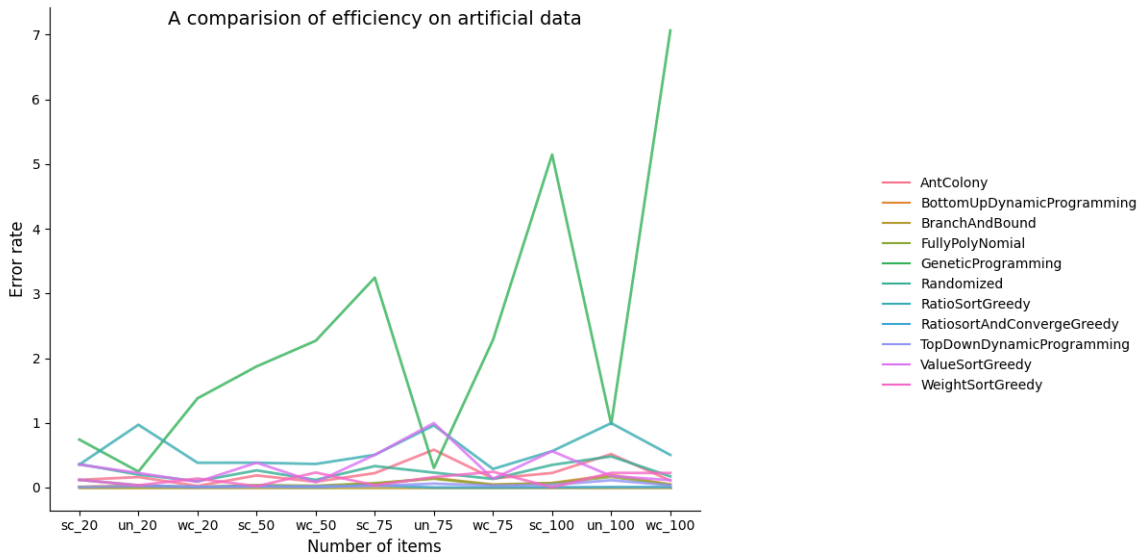

Figure 22: A comparison of efficiency on artificial data among all algorithms

the algorithm try all the possible way to sort the object in each sack. Let's suppose we have the sacks 1 to s. Then each object can be in the sack j $\forall j \in [\![1; s]\!]$.

Complexity So there is for each sub-part of the data-set s^k with k the number of object in the answer.
The total number of answer is, thanks to the Newton Binomial Law

$$\sum_{k=0}^{n} C_n^k * s^k = (s+1)^n$$

and so the time complexity of the algorithm is $O(n*(s+1)^n)$ with s the number of sacks and n the number of object.

4.1.2 Bottom-up Dynamic Programming

Algorithm Principle The principle of bottom-up dynamic programming in 0/1 multiple knapsack problem is the same as the one in 0/1 knapsack problem. So instead of applying the dynamic approach on one sack, we apply it to at most all the sacks. Basically, for each sack, we push in the items that can enter. If all the objects have already been pushed into the sacks before reaching the last sack, we stop the loop.

Complexity The time complexity of the procedure is $O(jnc_i)$ where j is the number of sack, n the number of items and c_i the highest weight among all the sacks weight.

4.1.3 Greedy Sort and Converge

Algorithm Principle It uses the same principles as 0/1 knapsack problem. When all the items are sorted, for each step, the algorithm try to put them one by one in the sack. There is no specific intelligence, the program is the following : For the current item, first it looks if its weight is equal to the free weight in each sack. If it is the case, it put this item in this sack and the sack will be considered as full. Second, if it didn't find the perfect place for the item, it will try to put it in one sack, whatever is the sack. When it find a sack with enough free weight, it put the item in the sack, and the sack's free weight is reduced.

The previous improvement of the algorithm, for 0/1 knapsack is not implement for multiple knapsack problem.

Complexity The complexity won't change comparing to 0/1 knapsack version, because, searching a sack for each item is $O(n * s)$ with s the number of sack. And sorting the item is $O(n * log(n))$. Supposing n much bigger than s, then the complexity won't change.
So the theoretical complexity is $O(n^4)$ and the real one should be $O(n * log(n))$ if and only if $logns$.

4.1.4 Multiple Randomized Algorithm

Algorithm Principle In the multiple version of the random algorithm, we will use the same insights that were applied for the single knapsack version. The main difference, is that instead of deciding at each iteration if we are going to add or remove and object, we will do it simultaneously in the multiple sacks provided.

For each iteration we will see for the c_j sacks if they are below or over their maximum weight. First, we check if there are sacks that exceeds their maximum weight condition. If a sack is over his value, we will get the objects that are currently included in it. In other words, the objects that are represented by an c_j for this particular sack (this will be explained lately). Then, we sort the objects by their value in ascending order, to pick one that does not reduce the total value for this sack meaningfully. Later on, we select from a small group of candidates, which it's size is defined by a parameter called selection ratio, a possible object randomly. This random selected object will be assigned a 0, which means that are not part of any sack.

Following this procedure, we will check if there are sacks that are below their conditions, which means that have extra space in them. So, now for this sacks, we will do the exact same procedure but taking into account the objects that are not in the current sack s and are free elements (have a 0 assignment). So, once

again we will choose among a reduce group the object that we will maximize the value for this specific sack.

It is important to notice that the encoding of the elements in the sacks can be done creating a list for each j $\in c_j$. However, this will not be a good option because an object can belong in two sacks at a certain moment. So, in order to fix this, we will control the object with a single list were each assignment is represented by 0 if it the object does not belong to any sack or the value j if the object $x_i \in c_j$. To illustrate this, let´s take the list $[0, 1, 1, 2]$. Assuming, that we have two sacks $(c_j, j = 2)$, the first element is free, the following two objects are included in the first sack and the last one belongs to the second sack.

Complexity The algorithm will stop when the number of iterations is reached or if the time condition is met. However, there is an extra element which generates more diversification in the algorithm. If at a certain point, the current solution of the iteration is valid (respects all maximum weights for all the sacks) and is the same as the previous iteration, means that we are in a local optima. So, we set a new random assignment to the objects, in order to explore other solutions. The complexity of the algorithm in O notation is the following O($s \cdot n \cdot log(n)$) where s is the number of sacks and n the number of objects to be assigned.

4.2 Experimentation

In order to test the different algorithms, we extracted 6 instances as benchmarks. Some instances don't have their optimal value specified. But with the help of the first and the three last data set for which we had the optimal combination of items, we had a rough idea of the quality of the algorithms. They were extracted from `https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_multiple/knapsack_multiple.html`. The instances have the structure shown in table 4.2.

4.2.1 Brute Force

1. Running time assessment For all the tests in figure 23, there was 10 elements in the knapsack, except the fifth one with only 6 elements. As we can see the number of sack has a huge impact in executed time. That make sense because the complexity is $O(n(s+1)^n)$ with s the number of sack. So for example 3 sacks with 10 objects have almost the same number of operation of 1 sack with 20 objects because $4^10 = 2^20$

2. Efficiency The efficiency is the best possible, because brute force take the best result between all the possible results.

4.2.2 Greedy Sort by ratio and Converge

1. Running time assessment I tested the algorithm with the 6 previous instances (see on **Table 4**) and the running time was less than 1 millisecond.

Instance Name	Elements	Sacks
instance_p01	10	2
instance_p02	10	3
instance_p03	10	4
instance_p04	10	1
instance_p05	6	2
instance_p06	10	2

Also, in order to avoid external factors, all the experiments below have been done using the same environment, which consisted in a Intel core i5 CPU with 3.10GHz of frequency, and 10 GB of RAM. Moreover, all executions were done independently.

Table 4: Multiple Knapsack Benchmarks

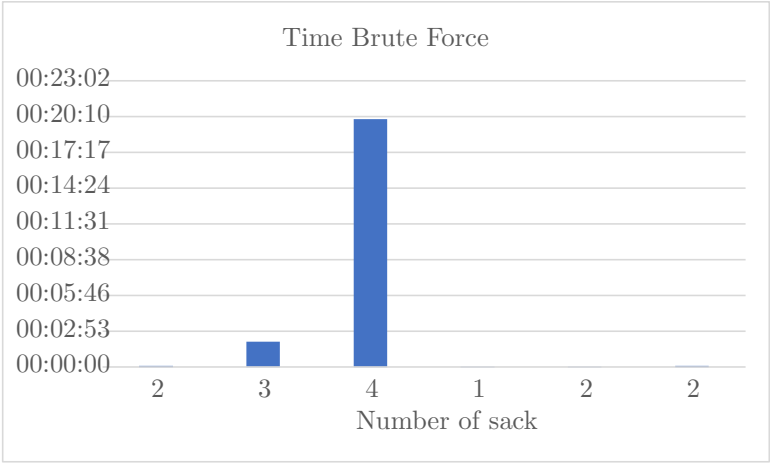

Figure 23: Executed time

2. Efficiency As we can see in the figure 24. This algorithm have quite bad results, with error rate around 5%.

3. Improvements It would be interesting to test the same improvement as with 0/1 knapsack. It is possible that it remove the huge error. But, a better way should be that the way to sort each item in each sack must be improved if we want better result.

Figure 24: Multiple Greedy - Error Rate

4.2.3 Bottom-up Dynamic programming

1. Running time assessment

Number of sacks	Weights of the sacks	Running time(us)	Value
1	165	1546	309
2	70,127	1479	333
2	65,85	584	345
2	103, 156	1428	451
3	50,81,120	1279	369
4	31,37,48, 152	393	1899 height

Table 5: Running time of Bottom-up dynamic programming on 0/1 Multiple knapsack problem

From the table above, we conclude that the running time of bottom-up dynamic programming increases with the number of sacks. This situation is due to the fact that we apply the dynamic programming approach on at most all the sacks.

2. Efficiency
Even though we didn't have a lot of data to provide a great analysis of the efficiency, we make the assumption that dynamic programming always provide the optimal value. So, we used the values returned by dynamic programming for running time as baselines for the running time assessment.

Number of sacks	Weights of the sacks	Error rate(1ms)	Error rate(10ms)	Error rate(100ms)
1	165	0	0	0
2	70,127	0	0	0
2	65,85	0	0	0
2	103, 156	0	0	0
3	50,81,120	0	0	0
4	31,37,48, 152	0.19	0	0 height

Table 6: Efficiency of Bottom-up dynamic programming on 0/1 Multiple knapsack problem

On those small instances, dynamic programming seems to be fast because it takes more than 1 second to complete its operations.

4.2.4 Randomized Algorithm

1. Running time assessment
   When comparing the different time executions, the algorithm manages to return the same solution among all the instances in a relative similar time. The only execution that performs poorly oscillates between 0.001 milliseconds and 1 milliseconds. This behavior shows that the convergence of the algorithm to return the same solution will not surpass 1 second for this 6 instances. Therefore, the time of execution above this time value will not improve the solution.

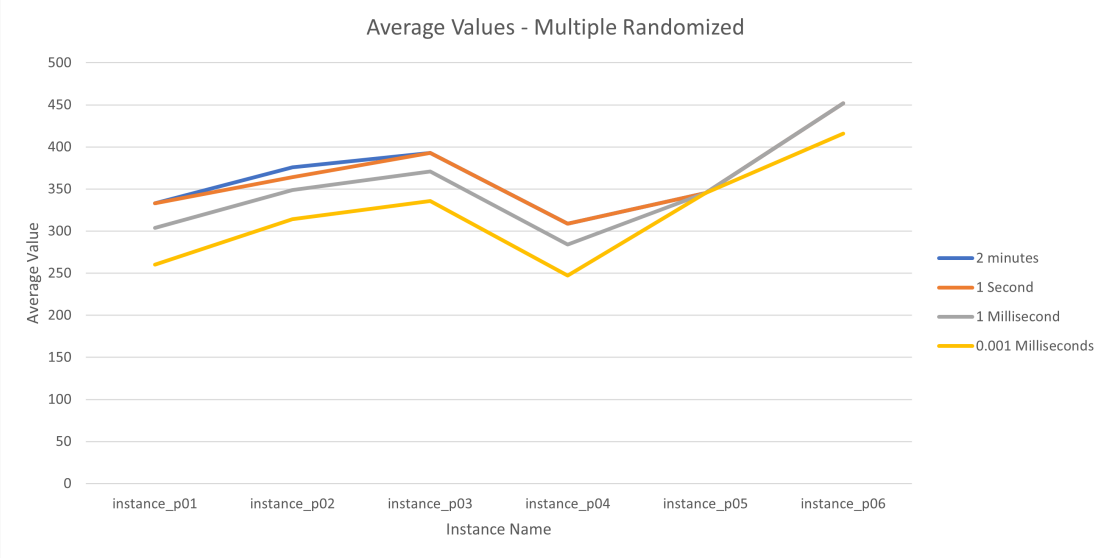

Figure 25: Average value with benchmarks - Multiple Randomized

2. Efficiency
   The algorithm performs best on instances p03 and p06, which have both 10 elements and 4 to 2 sacks respectively. This means, that in low dimensional instances it will be better to take less sacks when working with this randomized approach. Furthermore, the quality of the solution will be defined by the time which the algorithm executes. However, this will be a limit, in this case above 1 second the executions will bring the same value.

3. Some important decisions made
   When developing the algorithm, there were some improvements compared to the single sack version. The reset of the solution was introduced, so the algorithm could explore more possible solutions. The element selection encoding was important to avoid repeated elements among the sacks. Finally, iterating through all the sacks was a good choice in order to modify plenty elements in one iteration. Regarding the parameter of condition selection, the 0.3 value performed the best among all the executions. This value is the same as the selected in the single sack version.

4. Improvements
   This algorithm can be improved adding a stopping criteria. It can be added in the same section as the reset condition. The problem is that we will stop at the first local optima encountered.

5 Conclusions

The different approaches to solve the 0/1 knapsack problem have strong and weak points. Among all the instances there will be some algorithms that will outperform the other when taking the time measure and other with the quality measure. At the end, it depends in how much time and computational resources are available to execute the algorithms. As well as the kind of instance that we are executing: number of elements, maximum weight condition, correlation between the objects, etc.

Among all solutions explored, for the low dimensional instances we find out that a good balance between quality of solution and time taken to execute was primordial. Therefore, the dynamic approach, fully polynomial algorithms were a good candidate to select. However, for the large dimension instances it would be better an algorithm that takes short time in their calculation but returns a good estimate like the different greedy approaches proposed (including the one made by our team).

Finally, adding multiple sacks let us shown that in terms of time and quality the randomized or the dynamic approach were good candidates. Moreover, that depending on the number of sacks some algorithms will perform better than the others.

6 Planning of the project

At the beginning of the project, the planning assigned to each one of the tasks were proposed as follow:

6.1 Provisional Planning

6.1.1 Milestone 1: Algorithm Implementations

For this milestone, the algorithms will be separated in two waves. In the first wave, each member will work in the implementation of a specific algorithm. The first wave will be between: October 21 to November 1:

Algorithm	**Member**
Branch and Bound Algorithm	Christiane Manuela AYO NDAZO'O
Greedy Algorithm	Landry BAILLY
Dynamic Programming	Chadapohn CHAOSRIKUL
Genetic Algorithm	Gloria ISEDU
Randomized Approach Algorithm	Felipe CORTES JARAMILLO

Table 7: Provisional - First Wave Algorithms

The second wave will be between the dates: 14 November to 22 November:

Algorithm	**Member**
Fully Polynomial Time	Christiane Manuela AYO NDAZO'O
Brute Force Algorithm	Landry BAILLY
Minimum Spanning Tree	Chadapohn CHAOSRIKUL
Greedy Algorithm	Gloria ISEDU
Ant Colony Algorithm	Felipe CORTES JARAMILLO

Table 8: Provisional - Second Wave Algorithms

6.1.2 Milestone 2 and 3: Evaluate Algorithm and Benchmarks, Conduct Experimental Study

In order to evaluate and compare the different algorithms with their running times and results qualities, the work will be as we saw in the previous milestone. However, there will be general meetings to discuss all the algorithms and start writing the report. Between 1 November to 14 November:

Between 12 November to 14 November: Meeting to discuss the results of the algorithms. Also, include the results in the project report.

Between 2 December to 7 December: Meeting to discuss the results of the algorithms. Also, include the results and conclude the study with the results found through the project.

Task	Member
Test on 0/1 generator: Branch and Bound Algorithm	Christiane Manuela AYO NDAZO'O
Test on 0/1 generator: Greedy Algorithm	Landry BAILLY
Test on 0/1 generator: Dynamic Programming	Chadapohn CHAOSRIKUL
Test on 0/1 generator: Genetic Algorithm	Gloria ISEDU
Test on 0/1 generator: Randomized Approach Algorithm	Felipe CORTES JARAMILLO

Table 9: Provisional - Testing First Wave

Task	Member
Test on 0/1 generator: Fully Polynomial Time	Christiane Manuela AYO NDAZO'O
Test on 0/1 generator: Brute Force Algorithm	Landry BAILLY
Test on 0/1 generator: Minimum Spanning Tree	Chadapohn CHAOSRIKUL
Test on 0/1 generator: Greedy Algorithm	Gloria ISEDU
Test on 0/1 generator: Ant Colony Algorithm	Felipe CORTES JARAMILLO

Table 10: Provisional - Testing Second Wave

6.2 Real Planning and Workload

However the provisional planning changed meaningfully with the tasks and times that each one of the member was going to do. Among those changes the minimum spanning tree was removed as an algorithm required in the experiment, new algorithms were added like the own implementation of greedy approach, the testing module and generator were designed and made, the addition of multiple knapsack algorithms, etc. All the real planning is include in table 11.

October 19 - November 6		
Task	**Member**	**Milestone**
Development 0/1 Instance Generator	Felipe CORTES JARAMILLO	Algorithm Inputs
Development Brute Force Algorithm	Landry BAILLY	Algorithm Implementation
Development Branch and Bound Algorithm	Christiane Manuela AYO NDAZO'O	
Development Dynamic Programming Top Down Version	Chadapohn CHAOSRIKUL	
Development Genetic Algorithm	Gloria ISEDU	
Development Randomized Algorithm	Felipe CORTES JARAMILLO	
November 7 - 24 November		
Development Own Algorithm Approach	Landry BAILLY	Algorithm Implementation
Development Fully Polynomial Time Algorithm	Christiane Manuela AYO NDAZO'O	
Development Dynamic Bottom Up Version	Chadapohn CHAOSRIKUL	
Development Three Greedy Algorithms	Gloria ISEDU	
Development Ant Colony Algorithm	Felipe CORTES JARAMILLO	
Addition of correlation instances - 0/1 Generator	Christiane Manuela AYO NDAZO'O	Algorithm Inputs
Development Testing module	Landry BAILLY	Testing Tools
Individual Testing: Brute Force, Own Algorithm	Landry BAILLY	Testing and analysis
Individiual Testing: Fully Polynomial, Branch and Bound	Christiane Manuela AYO NDAZO'O	
Individiual Testing: Greedy and Genetic Algorithms	Gloria ISEDU	
Individual Testing: Randomized and Ant Colony	Felipe CORTES JARAMILLO	
25 November - 7 December		
Comparison of all algorithms: Time	Felipe CORTES JARAMILLO	Testing and analysis
Comparison of all algorithms: Quality of Solution	Chadapohn CHAOSRIKUL	
Comparison of all algorithms: Time	Landry BAILLY	
Comparison of all algorithms: Quality of Solution	Gloria ISEDU	
Comparison of all algorithms: Correlation in Instances	Christiane Manuela AYO NDAZO'O	
Development and Testing: Multiple Knapsack Randomized	Felipe CORTES JARAMILLO	Algorithm Implementation
Development and Testing: Multiple Knapsack Dynamic	Christiane Manuela AYO NDAZO'O	
Development and Testing: Multiple Knapsack Brute Force	Landry BAILLY	
GUI Design and inclusion analysis	Gloria ISEDU	Interface

Table 11: Real Planning

In the real planning we decided to write in the report as soon as we were getting the different algorithms implementations and the tests to each one of them. Mainly, because with the time constrain it was not feasible to leave all the writing at the end. Some tasks required more time than previously planned like the different testings made to the algorithms and the inclusion of elements that were out of the scope: testing module and generator correlation. However, the addition of these tasks were compensated with other activities to the other member of the group. Furthermore, as the different algorithms of the 0/1 knapsack were finished, the group included new algorithms of multiple knapsack, developing just few of the possible options. Furthermore, some elements that were in develpment did not work out at the end like the GUI proposed. However, some time of the project was used in their designing and thinking.

6.3 Workload

The workload among the group remained constant during all the project. When a new element appeared that was not expected, the other members of the team focused in other tasks. For example, when one member was implementing the testing module, two members focused in the remaining algorithms to implement and the remaining two in the tests that were needed to be explained more in the report. Also, the new extra elements that appeared were distributed equally among the team, like: generator, test module and overall comparison of the algorithms. With that explained the final workload percentage of work correspond to **20%** to each member of the team.

7 Report Checklist

1. **Did you proofread your report?** The report was read by each one of the members after it was done editing. Therefore, each one could detect any possible correction.

2. **Did you present the global objective of your work?** In the introduction of the report the global objective was mentioned and developed throughout all the report.

3. **Did you present the principles of all the methods/algorithms used in your project?** All principles were explained in the description of each algorithm, including important formulas, theory and expected time complexity.

4. **Did you cite correctly the references to the methods/algorithms that are not from your own?** All bibliographical cites were included in the report references. Furthermore, they were included in the explanation of several parts in the report: introduction and algorithm descriptions.

5. **Did you include all the details of your experimental setup to reproduce the experimental results, and explain the choice of**

the parameters taken? In each experimentation, the parameters for the execution were explained if it was the case. Also, for the group comparison, it is mentioned the conditions (parameters, time and environment) used in each tests. So, the results can be reproducible.

6. **Did you provide curves, numerical results and error bars when results are run multiple times?** Different tables and graphs were provided in the report, in order to explain better the behavior of the different algorithms. Among them, we have graphs that explain the error rates, time of average executions and average values.

7. **Did you comment and interpret the different results presented?** Each one of the algorithms were analysed in the experimentation section, which include an interpretation of the time, quality of the solution, strong points and possible improvements.

8. **Did you include all the data, code, installation and running instructions needed to reproduce the results?** All the code, data and installation are contained in the repository which contains all the project itself. In this repository the data are contained in the input folders, the code in the algorithm folders and finally, the installation in the repository description.

9. **Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/ techniques and debugging?** All the programs that contain the algorithms were designed to be encapsulated and separated between them. So, it would be easy to add a new method. Also, the input, output and main file can be used across all the algorithms, so the project can increase without problem. For debugging, the test module was made in order to execute all the programs in one single point, so it is easier to debug.

10. **Did you make sure that the results different experiments and programs are comparable?** All the experiments were made in the same environment, with the same resources. Also, the individual and multiple algorithms comparison, all the tests parameters were established the same across the algorithms, so the results will not be influenced by external factors.

11. **Did you sufficiently comment your code?** The code was commented properly, including an overall description of the program, references and individual comments to each function included in the files.

12. **Did you add a thorough documentation on the code provided?** The code was properly documented and commented as mention in the last question. Moreover, it was included a description in the project repository in order to explain the project in general.

13. **Did you provide the additional planning and the final planning in the report and discuss organization aspects in your work?** Yes, the provisional and final planning were added in the report. Also, some key points were explained, regarding the differences between the two plannings.

14. **Did you provide the workload percentage between the members of the group in the report?** The workload percentage is included in the real planning section. An explanation of the final percentage is provided.

15. **Did you send the work in time?** The report and the code were sent before 7 December 22:00 as the project description highlighted.

References

[1] Azzam Sleit Ameen Shaheen. Comparing between different approaches to solve the 0/1 knapsack problem. *IJCSNS International Journal of Computer Science and Network Security*, 16(7):1–10, 2016.

[2] Haiming Du, Zaichao Wang, Wei Zhan, and Jinyi Guo. Elitism and distance strategy for selection of evolutionary algorithms. *IEEE Access*, 6:44531–44541, 2018.

[3] Amaury Habrard. Greedy algorithms. `https://claroline-connect.univ-st-etienne.fr/web/app.php/resource/open/file/543747`, 2022. The author's organization: Laboratory Herbert Curien, Jean Monnet University.

[4] Hans Kellerer, Ulrich Pferschy, and David Pisinger. *Introduction to NP-Completeness of Knapsack Problems*, pages 483–493. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[5] 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.

[6] Nitish Kumar. Tabulation vs memoization. `https://www.geeksforgeeks.org/tabulation-vs-memoization/`, 2021.

[7] Prosun Kumar Sarkar. 0-1 knapsack problem using memoization. `https://www.geeksforgeeks.org/0-1-knapsack-problem-dp-10/`, 2022.

[8] KRZYSZTOF SCHIFF. Ant colony optimization algorithm for the 0-1 knapsack problem. 2013.

[9] Leon Ladewig Susanne Albers, Arindam Khan. Improved online algorithms for knapsack and gap in the random order model. 2021.

[10] Satvik Tiwari. Genetic algorithm: Part i - intuition. https://medium.com/koderunners/genetic-algorithm-part-1-intuition-fde1b75bd3f9, 2019.