Report

Advanced algorithms project

Authored by
Idriss BENGUEZZOU
Ghilas MEZIANE
Mohammed BOUTOUIZERA

Under Supervision of
Eduardo Brandao
Amaury Habrard

01/12/2022

Table of contents

1 Introduction

1.1 Overview

This report discusses the result of the work done in development of a solving knapsack problems application. this project is part of the "Advanced Algorithmics" module that is part of the 1st semester of our Data and Connected Systems master's degree. This project is a joint work with our colleagues from the Machine learning and data learning course.

1.2 Objective

The objective of this project is to implement some algorithms for solving the Knapsack Problem and to provide an experimental study of their running time and quality. Indeed,this project aims to evaluate the running time of the algorithms and the quality of the solutions given (in terms of optimality)

1.3 Composition of the group

Our group is composed of five members :

BENGUEZZOU Idriss, MEZIANE Ghilas, BOUTOUIZERA Mohammed,ROUABAH Mohammed, ZEGHDALLOU Ilyes. Each of us is part of the same master track (DSC : Données et systèmes connectés).

2 Setup

2.1 Technologies used

2.1.1 Language and IDE

For this project, we opted for the Python language. As a development environment, we decided to work on JupyterLab. The choice of the language is motivated by the fact that Python is a language that perfectly matches data-oriented projects. Regarding the IDE, our choice is due to the many features offered by Jupyterlab, which works with notebook files (.pynb), in which we can group both our code snippets and our experimental part.

We provided a gantt diagramme at the begining of this project, we now can compare the differences between the first diagramme and the current. As for why we went off schedule, a lack of involvement from two members led to a bigger load of work on the rest of group. We will detail the part realised by each member in the checklist at the end of this document.

2.1.2 Project management

As far as project management is concerned, we adopted the kanban method in order to distribute the tasks, and set deadlines.We also used the git tool to make it easier for us to work in collaborative mode.

2.1.3 Experimental setup

To be the more accurate possible when running our test we decided to launch the test on the same laptop the whole semester, and we tried to run the test on the same condition, so when running the test the laptop should only have one program opened (the IDE), and should not be used during the phase of test. The stats of the laptot are as follows :

- **Proccessor** : 2.50 Ghz
- **RAM Memory** : 12 Go
- **SSD Disk** : 237 Go

3 Implementation

3.1 Reminder of the knapsack problem

The knapsack problem is a problem in combinatorial optimization : Given a set of items, each with a weight and a value, determine the number of each item to include in a collection so that the total weight is less than or equal to a given limit and the total value is as large as possible. It derives its name from the problem faced by someone who is constrained by a fixed-size knapsack and must fill it with the most valuable items. The problem often arises in resource allocation where the decision-makers have to choose from a set of non-divisible projects or tasks under a fixed budget or time constraint, respectively.

In this project we will study 10 algorithms :

- The brute-force approach exploring all the possible solutions in a systematic way.
- Branch and bound approach
- Three greedy approaches
- The dynamic programming approach seen in the exercise sheet, and the top-down version that optimizes the number of calls
- The Fully polynomial time approximation scheme
- The optimal genetic approach and the non optimal approach)
- A Randomized approach
- A Genetic approach, with Ant Colony version

3.2 Source of the data

Before moving on to the implementation of the algorithms in order to solve the problem(s), we need to generate these problems.For this, we have two options :

1. We developed several types of generators with the help of mathematical functions in order to obtain different data distributions.
2. We developed a function that permits us to upload data-sets files, read them, fetch the data we need and fill the objects with it.

3.2.1 Generators

In order to represent our objects in the best possible way, we have defined a class $Item$, which has the two integer attributes : value and weight. Then, we use an array of items whose size is *nb_items* (the number of items) to store all our objects.

3.2.1.1 Random Generator We use the *randInt* function of python to generate a random integer for the value and the weight of each item. And then we fill our $choosed_items$ array.

3.2.1.2 Exponential Generator We use the exponential function to generate a random integer for the the weight of each item. By this we can ensure that our distribution is following an exponential path.

3.2.1.3 Normal law Generator In the same logic as the exponential generator, We used the normal law to get a normal distribution for our items.

3.2.2 Uploaded Datasets

In order to ensure efficient tests, and interesting experimental results, we took the initiative to implement a function that retrieves data from the dataset files proposed by our project tutor. So to summarize, the user has the choice between generating their data via the application or uploading them via the files belonging to the folder named "dataset".

3.3 Algorithms

3.3.1 Brute-force algorithm

Brute force is a straightforward approach to solving a problem, usually directly based on the problem's statement and definitions of the concepts involved.
Using brute force to solve the Knapsack problem is a simple solution. There will be 2n potential combinations of elements for the knapsack if there are n elements to pick from. An element is either selected or not. A bit string of 0's and 1's with a length equal to the number of elements n, is created. If the i^{th} symbol in a bit string is 0, the element is not selected. And if the answer is 1, the element is picked.
Time complexity of the algorithm : $O(2^n)$ when n is the number of items.

3.3.2 Dynamic programming algorithm

Dynamic Programming is an algorithmic technique for solving an optimization problem by breaking it down into simpler subproblems and utilizing the fact that the optimal solution to the overall problem depends upon the optimal solution to its subproblems.
To solve 0/1 knapsack using Dynamic Programming we construct a 2D array having as dimensions :

$$[nb\_items + 1]\ [capacity + 1]$$

The rows of the table correspond to items from 0 to nb_items. The columns of the table correspond to weight limit from 0 to *capacity*. So the very last cell of our array is in the position $[nb\_items + 1]$ $[capacity + 1]$. The value attribute of the cell in position [i][j] represents the maximum profit possible when considering items from 0 to i and the total weight limit as j. So after filling the table, we can say that our solution (maximal profit) is the value of the cell which is in position $[nb\_items + 1]$ $[capacity + 1]$.

So how do we fill the table ?

First of all, we set the 0th row and column to 0. We do this because the 0th row means that we have no items and the 0th column means that the maximum weight possible is 0. Now,for each cell [i][j],we have the choice between including or not the object at the position [i] in our final selection. To include the item at the position [i], the total weight after including item [i] should not exceed the weight limit , and the profit after should be greater as compared to when item [i] is not included. By applying this rule for each cell, we finish to fill our table and we can fetch the last cell to deduce the solution. **Time complexity of the algorithm :** $O(n * w)$ when n is the number of items and w is the capacity.

3.3.2.1 Top-Down approach

It is based on memorization techniques. This approach divides a large problem into multiple sub-problems and solves sub-problems only if the solution is not memorized. That is, there is no need to recalculate the same subproblems.

3.3.3 Fully polynomial time scheme

Polynomial Time Approximation Scheme (PTAS) is a type of approximate algorithms that provide user to control over accuracy which is a desirable feature. These algorithms take an additional parameter > 0 and provide a solution that is $(1 + \epsilon)$ approximate for minimization and $(1 - \epsilon)$ for maximization. Fully Polynomial Time Approximation Scheme (FPTAS) is a stricter scheme of PTAS. In FPTAS, algorithm need to be polynomial in both the problem size n and 1. In this algorithm, we firstly find the maximum valued item, i.e. the maximum $choosed\_items[i].value$ Let this maximum value be maxVal.
Then we compute adjustment factor k for all values, with :

$$k = (maxVal * \epsilon)/n$$

And finally we adjust all the values by creating a new array. This new array with adjusted values is passed as a parameter to the dynamic programming algorithm, which will calculate our solution. [5]

$$val'[i] = floor(val[i]/k)$$

3.3.4 Greedy

Greedy programming techniques are used in optimization problems. They typically use some heuristic or common sense knowledge to generate a sequence of suboptimum that hopefully converges to an optimum value [3].
There is 3 posiibilities of greedy algorithm

3.3.4.1 Greedy Ratio
Choose the items with as high a value per weight as possible(that mean we have to sort items by the ratio weight value in in descending order).

3.3.4.2 Greedy Weight
Choose the items with less weights as possible(that mean we have to sort items by the weight in in ascending order).

3.3.4.3 Greedy Value
Choose the items with highs values as possible(that mean we have to sort items by the value in in descending order).

Fractional knapsack problem The fractional knapsack problem also uses greedy method based on ratio, it's maximise the value and take all weight, but it's not considered as 0/1 knapsack problem (In fractional we can break items) That is not authorised in 0/1 knapsack problem.

Time complexity of the algorithm

1. Sorting items **O(NlogN)**
2. choose items **O(N)**

$\mathbf{O(NlogN)} + \mathbf{O(N)} \approx \mathbf{O(NlogN)}$

3.3.5 Branch and Bound

Branch and bound is an algorithm design paradigm which is generally used for solving combinatorial optimization problems. These problems typically exponential in terms of time complexity and may require exploring all possible permutations in worst case. Branch and Bound solve these problems relatively quickly. It creates and enumerates branches of possible solutions in order to pick the optimal solution. Regions of the tree where there is no solution are deleted consecutively. After exploring the whole tree, a solution is proposed. This algorithm is used for optimization purposes.

3.3.6 Markov Chain Monte Carlo (MCMC)(Randomized)

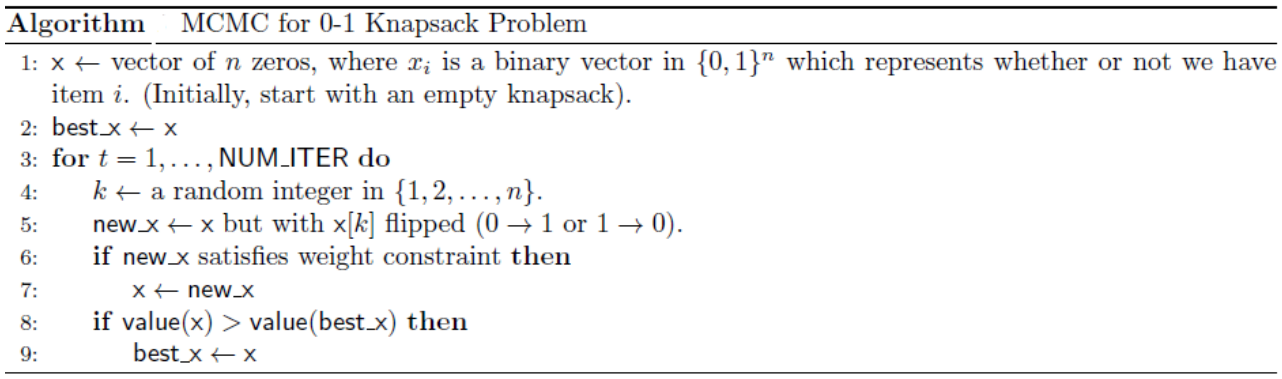
3.3.6.1 Markoc chaine
or Markov process is a stochastic model describing a sequence of possible events in which the probability of each event depends only on the state attained in the previous event [2].

3.3.6.2 Markov Chain Monte Carlo (MCMC)
is a technique which can be used to hard optimization problems (though generally it is used to sample from a distribution). The general strategy is as follows :

— Define a Markov Chain with states being possible solutions, and (implicitly defined) transition.

— Run MCMC (simulate the Markov Chain for many iterations until we reach a good state of solution)

The states will be all possible solutions : binary vectors x of length n (only having 0/1 entries). We'll then define our transitions to go to good states (ones that satisfy our weight constraint), while keeping track of the best solution so far. This way, our stationary distribution has higher probabilities on good solutions than bad ones. Hence, when we sample from the distribution (simulating the Markov chain), we are likely to get a good solution !



Algorithm MCMC for 0-1 Knapsack Problem

```
1: x ← vector of $n$ zeros, where $x_i$ is a binary vector in $\{0,1\}^n$ which represents whether or not we have
   item $i$. (Initially, start with an empty knapsack).
2: best_x ← x
3: for $t = 1, \dots,$ NUM_ITER do
4:     $k$ ← a random integer in $\{1, 2, \dots, n\}$.
5:     new_x ← x but with x[$k$] flipped ($0 \to 1$ or $1 \to 0$).
6:     if new_x satisfies weight constraint then
7:         x ← new_x
8:     if value(x) > value(best_x) then
9:         best_x ← x
```

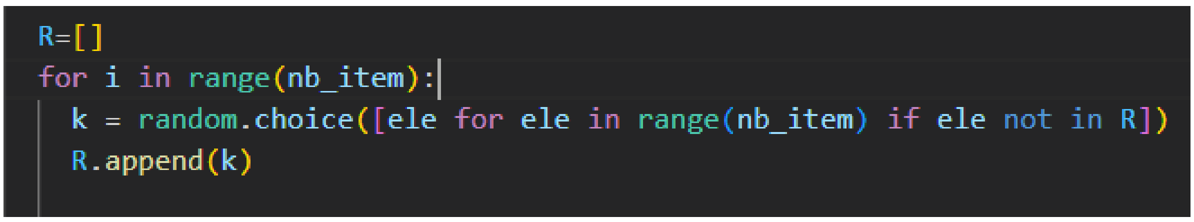
FIGURE 1 – CMCM

Basically, this algorithm starts with the guess of x being all zeros (no items). Then, for NUMITER steps, we simulate the Markov Chain. Again, what this does is give us a sample from our stationary distribution. Inside the loop, we literally just choose a random object and flip whether or not we have it. We maintain track of the best solution so far and return it.
So this MCMC definitely won't guarantee us to get the best solution, but it leads to **Dumb** solutions that actually work quite well in practice [4].

Note This is just one version of MCMC for the knapsack problem, there are definitely probably better versions and that what we prouved by updating this approch and make one better.

3.3.7 Personnel version based on Randomized approach

We take the same previous algorithm (CMCM) that is based on Markov chains and randomize choises, just we changed the way used to generate the random numbers. Our appraoch can generate random numbers with a simple way, the difference is that we can't generate the same number twice so we don't need to flip (0 to 1 or 1 to 0) and we guarantee to visit all items and one time unlike previvous algorithme (generate one number multiple time and flip each time, that mean take items **state 1** and return them **state 0**) .



```python
R=[]
for i in range(nb_item):
    k = random.choice([ele for ele in range(nb_item) if ele not in R])
    R.append(k)
```

FIGURE 2 – CMCM Own version

Time complexity of the algorithm O(N^2)

3.3.8 Genetic algorithms (GAs)

Genetic algorithms are probabilistic search algorithms that mimic biological evolutionary processes, allowing users to solve complex optimization problems. They work based on a population of chromosomes, with one chromosome representing a candidate solution. The genetic operator causes the population to evolve over time and converge towards an optimal solution.
Of course, in the spirit of "survival of the fittest", GAs are designed to solve maximization problems where chromosomes with higher fitness are more likely to survive and the objective function is called the fitness function. A fitness function is a mathematical formula that assesses the "quality" of a chromosome and directs evolutionary searches for regions of good fitness in a given search space.

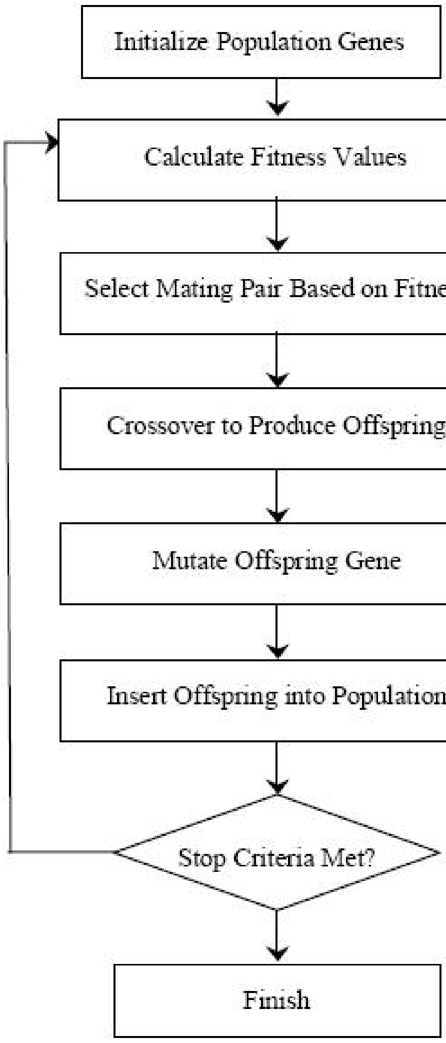

Initialize Population Genes
Calculate Fitness Values
Select Mating Pair Based on Fitness
Crossover to Produce Offspring
Mutate Offspring Gene
Insert Offspring into Population
Stop Criteria Met?
Finish

Figure 3 – Design of Genetic Algorithm

Given the zero-one encoding scheme, SGA is of course applicable to solving the knapsack problem. The objective of the knapsack problem like we said before is to maximize the total profit (p) of selecting items.Each with a profit p_i and a weight w_i, from a given item set of size n, while simultaneously preventing the violation of the constraint that the total weight of selected items exceeds the knapsack's capacity W.

Let $f()$ represent the fitness function and $x = (x_1, \ldots, x_n)$ be a binary vector, indicating selected items, i.e., $x_i = 1$ if the ith item is selected into the knapsack, and $x_i = 0$ otherwise. The knapsack problem can be formally described as follows :

$$mazimise\{f(x)\} = maximise\{\sum_{i=1}^{n} x_i p_i\}$$

$$\text{subject to } \{\sum_{i=1}^{n} x_i w_i < w\}$$ [1]

4 Experimentation & analyzing

After we implemented our algorithms, we focused on testing to obtain some analyzable results. First of all we decided to analyze the time of execution and the memory usage for each algorithm, and this by testing on different datasets : Random generated dataset , Normal distribution generated dataset and an external dataset. In order to increase the accuracy of our tests, we decided to launch every algorithm 10 times, so that we have 10 execution times. We will take on every algorithm the average of the 10 times obtained.

For each algorithm we provide curves and tables inside the pdf files that are joined with this report, we give the references of these documents inside all the sections below. The curves and the graphs are in the folder results_data/analysis/all

4.1 Brute Force

We tested this algorithm only on a small scale dataset (lesser than 100 items), due to its complexity, and the limits of our machines.

References :
Random generator :
analysis_random_small.pdf- Brute Force

Normal distribution generator : analysis_normal_small.pdf- Brute Force
analysis_normal_medium.pdf- Brute Force

We notice from these graphs that the execution time increases dramatically when the number of items increases. This can be explained by the complexity of the brute force algorithm, which tests all possible combinations. We can already guess that it is not the fastest algorithm. We also can see that the execution time increases as we increase the capacity, this result can be explained by our implementation, indeed we search the best value and combination thus the algorithm must compute more data.

4.2 Dynamic programming

References :
Random generator :
analysis_random_small.pdf- Dynamic programming

analysis_random_medium.pdf- Dynamic programming
analysis_random_large.pdf- Dynamic programming

Normal distribution generator :
analysis_normal_small.pdf- Dynamic programming
analysis_normal_medium.pdf- Dynamic programming
analysis_normal_large.pdf- Dynamic programming

External dataset :
analysis_dataset_medium.pdf- Dynamic programming

From these graphs, we notice that the growth of the execution time is linked to the growth of the capacity, and this, whatever the type of input data is. However, we note that the execution time is much lower than the bruteforce algorithm. Concerning the solution proposed by the algorithm, Dynamic programming approach is efficient since it always finds the optimum. For example in 3 tests for a medium-scale external dataset, the optimum is finded, for different capacities (995, 1008, 2543), and different quantities of items (100, 200, 500 items).

4.3 Dynamic programming top and down version

References :
Random generator :
analysis_random_small.pdf- Dynamic programming TOP DOWN
analysis_random_medium.pdf- Dynamic programming TOP DOWN
analysis_random_large.pdf- Dynamic programming TOP DOWN

Normal distribution generator :
analysis_normal_small.pdf- Dynamic programming TOP DOWN
analysis_normal_medium.pdf- Dynamic programming TOP DOWN
analysis_normal_large.pdf- Dynamic programming TOP DOWN

External dataset :
analysis_dataset_medium.pdf- Dynamic programming TOP DOWN

From these graphs, we can make the same conclusion as for the other algorithms concerning the increase of the time related to the increase of the capacity. We can also say that the top and down version is slower than the dynamic programming regarding to the average execution time. For the normal distribution tests, we can explain the graphs by the fact that , When the algorithm arrives to the biggest weights, the time execution increases. and that's logical when we compare it to the normal distribution graph.

4.4 The Fully polynomial time approximation scheme (FPTAS)

References :
Random generator :
analysis_random_small.pdf- Dynamic programming FPTAS
analysis_random_medium.pdf- Dynamic programming FPTAS

analysis_random_large.pdf- Dynamic programming FPTAS

Normal distribution generator :
analysis_normal_small.pdf- Dynamic programming FPTAS
analysis_normal_medium.pdf- Dynamic programming FPTAS
analysis_normal_large.pdf- Dynamic programming FPTAS

External dataset
analysis_dataset_medium.pdf- Dynamic programming FPTAS

Whatever dataset is provided to this algorithm, the execution time grows linearly with the capacity and the number of items. However, the FPTAS algorithm seems to be slower than the top and down algorithm. For example, for a capacity of 1000 and 1000 items as input, the FPTAS takes approximatively 4.1 secondes to execute , while the top and down version, takes only 0.90 secondes. the FPTAS algorithm has approximatively the same execution time as the classical dynamic programming algorithm.

4.5 Greedy weight

References :
Random generator :
analysis_random_small.pdf- Greedy weight
analysis_random_medium.pdf- Greedy weight
analysis_random_large.pdf- Greedy weight

Normal distribution generator :
analysis_normal_small.pdf- Greedy weight
analysis_normal_medium.pdf- Greedy weight
analysis_normal_large.pdf- Greedy weight

External dataset
analysis_dataset_medium.pdf- Greedy weight

When we provide the algorithm with data generated with the normal distribution generator, the algorithm decrease its time of execution as we increase the capacity, this can be explained by the normal distribution curve that generate symetric data. The greedy weight algorithm sort the items by their weights and then choose the item that fit in the knapsack, that's why the execution time decrease. When we test the same algorithm with a random generated dataset, the execution time also decrease but not as significantly as when we generated the data with the normal distribution. The graph also seems to reach a maximum. We even tested the algorithm with an external dataset, wich when the result put on a chart describe an increase of the execution time when the number of items and capcity is increased. We can notice that the execution time increase rapidly when the number of items is increased.

4.6 Greedy value

References :
Random generator :
analysis_random_small.pdf- Greedy value
analysis_random_medium.pdf- Greedy value
analysis_random_large.pdf- Greedy value

Normal distribution generator :
analysis_normal_small.pdf- Greedy value
analysis_normal_medium.pdf- Greedy value
analysis_normal_large.pdf- Greedy value

External dataset
analysis_dataset_medium.pdf- Greedy value

The greedy value algorithm has the same pattern when observing the generated curve, that's logical the sole difference in the algorithm is the sorting that is applied to the values. When observing the result we noticed that the greedy value algorithm provide a better solution than the greedy weight.

4.7 Randomized approach (CMCM)

References :
Random generator :
analysis_random_small.pdf- RANDOM CMCM
analysis_random_medium.pdf- RANDOM CMCM
analysis_random_large.pdf- RANDOM CMCM

Normal distribution generator :
analysis_normal_small.pdf- RANDOM CMCM
analysis_normal_medium.pdf- RANDOM CMCM
analysis_normal_large.pdf- RANDOM CMCM

External dataset
analysis_dataset_medium.pdf- RANDOM CMCM
For CMCM since it's randomize algorithm , it will never give optimal soltution and that appears in table of comparaison, it's always far than optimal solution and the execution time increase as we grow the number of items.

4.8 Genetic No optima

References :
Random generator :
analysis_random_small.pdf- Genetic No optima
analysis_random_medium.pdf- Genetic No optima

Normal distribution generator :
analysis_normal_small.pdf- Genetic No optima

External dataset
analysis_dataset_medium.pdf- Genetic No optima

The genetic algorithm is launched with two parameter the numbers of genome we want to create each generation and the number of generation. Thus the time execution will mainly depends on these two parameter, indeed for the curve provided in the analysis document, we can see that watever the capcity or the number of items the execution time is the same. But when we change the number of generation or the number of genome the execution time grow exponentialy.

4.9 BRANCH AND BOUND BFS

References :
Random generator :
analysis_random_small.pdf- BRANCH AND BOUND BFS
analysis_random_medium.pdf- BRANCH AND BOUND BFS
analysis_random_large.pdf- BRANCH AND BOUND BFS

Normal distribution generator :
analysis_normal_small.pdf- BRANCH AND BOUND BFS
analysis_normal_medium.pdf- BRANCH AND BOUND BFS
analysis_normal_large.pdf- BRANCH AND BOUND BFS

External dataset
analysis_dataset_medium.pdf- RANDOM BRANCH AND BOUND BFS

4.10 BRANCH AND BOUND DFS

References :
Random generator :
analysis_random_small.pdf- BRANCH AND BOUND DFS
analysis_random_medium.pdf- BRANCH AND BOUND DFS
analysis_random_large.pdf- BRANCH AND BOUND DFS

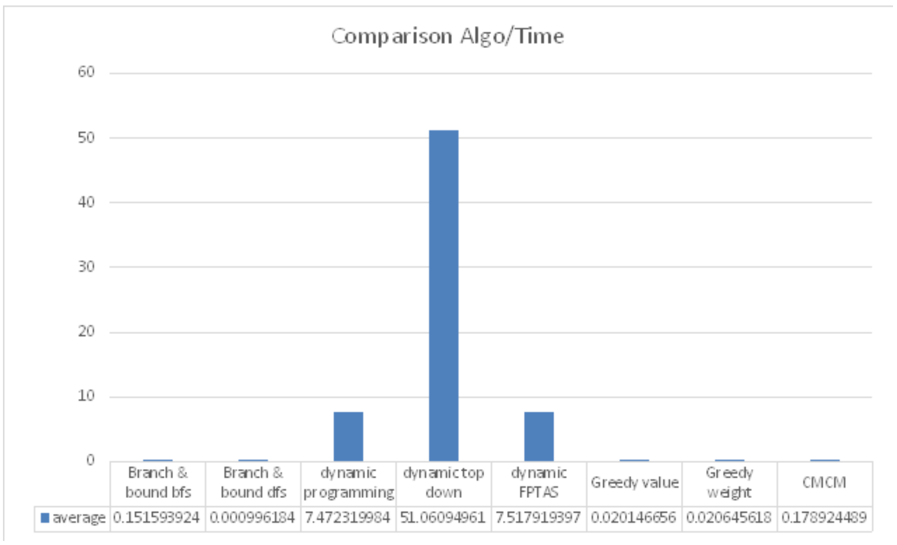
Normal distribution generator :
analysis_normal_small.pdf- BRANCH AND BOUND DFS
analysis_normal_medium.pdf- BRANCH AND BOUND DFS
analysis_normal_large.pdf- BRANCH AND BOUND DFS

External dataset
analysis_dataset_medium.pdf- RANDOM BRANCH AND BOUND BFS

5 General comparison

5.1 Time comparison

After studying each algorithm on different datasets, we can conclude that there is a strong corellation between the number of items, and execution time whatever distribution used. If we compare between the algorithms, regarding the time, using the same dataset, we can deduce that the slowest algorithms are : Dynamic Programming approaches, Brute and force, Genetic algorithm. While the greedy algorithms seems to be really fast.



	Branch & bound bfs	Branch & bound dfs	dynamic programming	dynamic top down	dynamic FPTAS	Greedy value	Greedy weight	CMCM
■ average	0.151593924	0.000996184	7.472319984	51.06094961	7.517919397	0.020146656	0.020645618	0.178924489

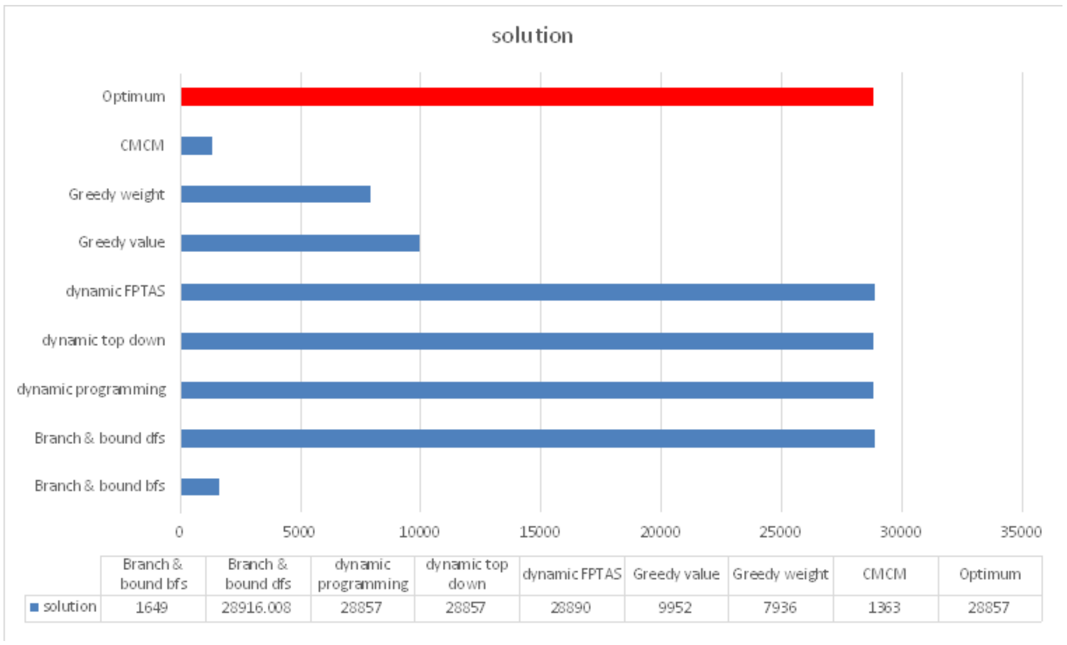
Figure 4 – Comparison on 500 items and 2543 capacity

5.2 Solution comparison

The efficency of the algorithm is defined by the solution it finds. Dynamic programming and brute force are always returning the optimal solution, even though they are the slower to execute. Greedy approaches are losing efficiency by gaining time, since they are returning a solution that is far from the optimal solution.

5.3 Memory usage

Our study allowed us to conclude that all the algorithms are similar in memory usage, except the Genetic algorithm, and the dynamic top and down.



	Branch & bound bfs	Branch & bound dfs	dynamic programming	dynamic top down	dynamic FPTAS	Greedy value	Greedy weight	CMCM	Optimum
■ solution	1649	28916.008	28857	28857	28890	9952	7936	1363	28857

FIGURE 5 – Comparison on 500 items and 2543 capacity

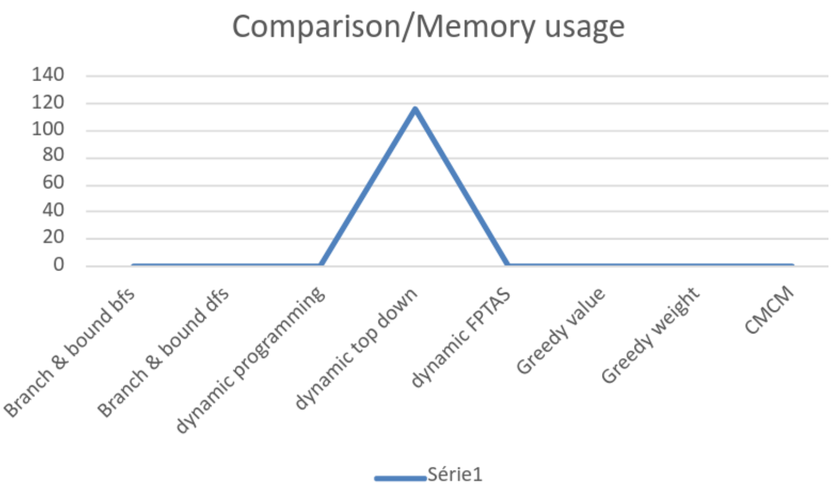


FIGURE 6 – Comparison on 500 items and 2543 capacity

6 Checklist

1. Did you proofread your report ? YES
2. Did you present the global objective of your work ? YES
3. Did you present the principles of all the methods/algorithms used in your project ? YES
4. Did you cite correctly the references to the methods/algorithms that are not from your own ? YES
5. Did you include all the details of your experimental setup to reproduce the experimental results, and explain the choice of the parameters taken ? YES
6. Did you provide curves, numerical results and error bars when results are run multiple times ? YES
7. Did you comment and interpret the different results presented ? YES
8. Did you include all the data, code, installation and running instructions needed to reproduce the results ? YES

9. Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/techniques and debugging ? YES
10. Did you make sure that the results different experiments and programs are comparable ? YES
11. Did you sufficiently comment your code ? YES
12. Did you add a thorough documentation on the code provided ? YES
13. Did you provide the additional planning and the final planning in the report and discuss organization aspects in your work ?
14. Did you provide the workload percentage between the members of the group in the report ? Here it is !

- BENGUEZZOU Idriss 30%
- MEZIANE Ghilas 30%
- BOUTOUIZERA Mohammed 30%
- ROUABAH Mohammed 7%
- ZEGHDALLOU Ilyes 3%

Initialization of the project : Benguezzou - Meziane - Boutouizera
implementation of the generators : Benguezzou - Meziane
Uploading and testing external datasets : Benguezzou Brute force Algorithm : Benguezzou - Meziane
Dynamic programming Algorithm : Benguezzou - Meziane Dynamic programming Top down Algorithm : Benguezzou - Meziane
Fptas algorithm : Boutouizera
3 Greedy approaches : Boutouizera
CMCM algorithm : Boutouizera
Branch and bound : Benguezzou - Meziane
Branch and bound DFS/BFS : Rouabah
Genetic algorithm : Benguezzou - Meziane
Ant colony algorithm : Zeghdallou
Testing functions : Benguezzou - Meziane
Analyze and Test : Benguezzou - Meziane - Boutouizera
Report : Benguezzou - Meziane - Boutouizera
Additional report : Zeghdallou Rouabah

References

[1] Scrucca, GA : A package for genetic algorithms in R, Journal of Statistical Software, vol. 53, no. 4, pp. 1–37, 2013.

[2] https ://en.wikipedia.org/wiki/Markov$_chain$

[3] Hristakeva, Maya. “Different Approaches to Solve the 0 / 1 Knapsack Problem.” (2005).

[4] https ://web.stanford.edu/class/archive/cs/cs109/cs109.1218/files/student$_drive/9.6.pdf$

[5] https ://www.geeksforgeeks.org/polynomial-time-approximation-scheme/