

Experimental Study Report : Knapsack Problem

Pierre BONNEFOY, Ophélie MARECHAL, Ian RIVIERE,
Alexandre MARINE, Aloys LANA

December 7, 2022

1 Abstract

The knapsack problem is a very common NP problem. For this reason, we try today to solve it using different approaches. Basically, the problem is set like this :

Given a list of objects such that an object has a profit p and a weight w . You are given a knapsack with a maximum weight capacity W .

The objective here is to maximize the profit of all objects that we can fit into the maximum capacity of the knapsack.

We solved this problem in 0/1 constraints, and for some algorithms in Multidimensional constraints.

To do this, we try all those different approaches :

- Ant colony
- Branch and bound
- Brute force
- Dynamic programming
- Fully polynomial time approximation scheme
- GRASP
- Greedy (all)
- Personal approach (meet-in-the-middle tweaked version)

Then we will try to study the different results of each approach in order to find which approach was the best considering three factors:

- Time complexity
- Space complexity
- Solution accuracy

Contents

1	Abstract	1
2	Introduction	2
3	Working Organisation	4
3.1	Tasks Repartition	4
3.2	Planification	5
4	Implementation Choices	5
4.1	Ant Colony	5
4.2	Branch and Bound	5
4.3	Brute Force	6
4.4	Dynamic Programming	6
4.5	Fully polynomial time approximation scheme	6
4.6	GRASP	6
4.7	Greedy	7
4.8	Personal approach	7
5	Tests	8
5.1	Test procedure	8
5.2	Test Results on 0/1 Problems	8
5.2.1	Time Complexity	8
5.2.2	Solution Accuracy	11
5.3	Test Results on Multidimensional Problems	11
5.3.1	Time Complexity	11
5.3.2	Accuracy	12
5.4	Results Interpretation	12
5.4.1	0/1 Problems	12
5.5	Multidimensional Problems	13
6	Workload	13
7	Conclusion	13
A	Annexes	14
A.1	Checklist	14
A.2	Algorithms	15

2 Introduction

In order to solve the knapsack problem, we first took the time to create a knapsack problem generator, capable of creating 0/1 knapsack problems, in order to grant us an infinite batch of files to work on. We also included some well-known databases to work on, namely :

For 0/1 knapsack problems.

- low-dimensional and large-scale from [Johny Ortega](#)

For Multidimensional problems.

- gk from [F. Glover and G. Kochenberger](#)
- chubeas from [P. C. Chu and J. E. Beasley](#)

You can find the requirements and launching instructions into the README.md file of the project wich is a Markdown Document.

3 Working Organisation

3.1 Tasks Repartition

Table 1: Table representing the repartition of the tasks for each member of the group.

Task	Name
Generator	Ian RIV-IERE
Terminal main	Alexandre MARINE
Interface	Pierre BON-NEFOY
Generation of analysis graphics.	Ophélie MARECHAL
Ant Colony fo 0/1 problems.	Pierre BON-NEFOY
Branch and Bound for 0/1 problems.	Ophélie MARECHAL
Brute Force for 0/1 problems.	Aloys LANA
Dynamic Programming for 0/1 problems.	Ian RIV-IERE
FPTAS for 0/1 problems.	Ian RIV-IERE
GRASP for 0/1 problems.	Alexandre MARINE
Greedy for 0/1 problems.	Alexandre MARINE
GRASP for Multidimensional problems.	Alexandre MARINE
Greedy for Multidimensional problems.	Alexandre MARINE
Brute Force for Multidimensional problems.	Aloys LANA

3.2 Planification

Table 2: Table representing the planification of the project and the status of the tasks.

Deadline	Task	State
21/10/2022	Writing the Group Description, Planification and Task Repartition.	Done
28/10/2022	Definition of the format used for the generator files. Developement of the Knaspack Problem Generator.	Done
10/11/2022	Each member will have implemented one method on 0/1 Knapsack Problems.	Done
17/11/2022	All algorithm are implemented on 0/1 Knapsack Problems	Done
30/11/2022	The study of algorithm is started and the Brute Force, Greedy, GRASP are implemented for Multidimensional Knapsack Problems.	Done
04/12/2022	Graphic Interface finished.	Done
07/12/2022	Study Report is finished and upload of the final version.	Done

4 Implementation Choices

4.1 Ant Colony

We choose to set two variables `MINPHERO` and `MAXPHERO` to 0.01 and 6 respectively as they are the registered values of this [study archive](#).

After some test from Pierre's on his own, he spotted that those parameters weren't really impactful in opposition to these next variables : `ALPHA`, `RHO` and `BETA`.

`ALPHA` corresponds to how important pheromones will be for ants.

`RHO` corresponds to the evaporation ratio of pheromones.

`BETA` corresponds to how important the ratio of objects profit and weight is for ants.

Those variables are set such that `ALPHA = 2`, `RHO = 0.98` and `BETA = 5` and were determined by the scientists of the study. After a lot of test from Pierre, he confirms that those values are the most optimized parameters for working with ants.

Only those five variables comes from the study alongside their method for computing the probability as Pierre already wrote the algorithm and was already getting some interesting results on all test files.

Then he chose to not implement the more accurate version of the study as the results granted with the ratio calculus were good enough.

4.2 Branch and Bound

For the branch and bound algorithm, in order to try to optimize the space complexity and the execution time, we decided that rather than creating an entire tree, the algorithm only consider the information concerning the current node (a set of objects, its value and weight) and the best solution found (a final set of objects, the final value and the final weight), which is equivalent to a leaf of the tree.

This way, the algorithm will browse the list of objects and for each of them, try to take it or not, which is similar to the brute force approach. But the particularity of the branch and bound algorithm is that if the best of the remaining objects compose a solution which is worse than

current best, the exploration of these items can be ignored, which allows a large calculation reduction.

The two set of objects are represented by tables which are filled with a 0 if the object at this index isn't taken, and 1 if it's taken.

4.3 Brute Force

The brute force algorithm is going through all the possible solutions so in order to try to optimize the execution speed, we made two choices :

- Generation of all the possibilities by using a list of all the N binary combinations, where N corresponds to the number of object of the problem. Each bit, corresponds to the status of the object : a value of 1 means that the object is in the Knapsack and a value of 0 means that it isn't in. For example, with $N = 2$ we obtain the following result : $[[0,0],[0,1],[1,0],[1,1]]$. This allow us to not going through all the Object List at each iteration.

- Reduction of the calculations, if the weight of the current possibilty is outrunning the Knapsack Capacity, we stop going through this possibility and we go through the next one.

- By the same idea, we reduce the number of calculation for Multidimensional Knapsack Problems, if the weight of the current possibilty of this dimension is outrunning the Knapsack Capacity of this dimension, we stop going trough this possibility and we go through the next one.

4.4 Dynamic Programming

The dynamic programming approach was implemented using our courses. This means that we divide the problem into smaller problems and we keep track of all values found until the end in a matrix created like that : $matrix[n][m]$ with n the number of items and m all possible weighth.

While checking for all objects if the object weight exceed the maximum capacity, we ignore it. But if it could fit, we check if it's value is of greater interest than the last one included. If yes, we then include it, otherwise we don't.

After this we built a Truth list to check which item was included.

4.5 Fully polynomial time approximation scheme

We were asked to implement a fully polynomial time approach, the easiest way for us was from upgrading the dynamic approach.

The dynamic approach has a big flaw, it runs way better on smaller value, and tends to slow down when the profit becomes bigger.

That's where the fptas approach shines the best, by reducing the profits to value between 0 and 9. We can speed up the dynamic algorithm by a lot. In order to do so, we check for the higher power of 10 objects in the list. After this we can build an auxiliary list of objects but with value restricted to the sets $\{0,1,2,3,4,5,6,7,8,9\}$.

However this should mean that we can't get all objects back from the dynamic processing (as all objects were reduced and we couldn't guess which object was the first with value 1 for example). In order to prevent that, we loose some space complexity optimization by creating a list of indexes of all objects included, so that we can find them way easier after the dynamic procees.

4.6 GRASP

For our random approach we choose to work with GRASP methodology. It consists of running a Greedy approach but in order to try to optimize the final value we take a random object

between x objects, x being a value that we choose. As it is random, the process will run t times the random version of greedy.

This allow us to change how accurate the algorithm can be, for instance we choose to set it to 3. This makes the algorithm run a little bit random as we choose randomly between 3 items, but it stills allow us to reach the optimal choice as it is between the 3 best choices at this time of the computation.

If we tweak this parameter to take more objects for the choice, we will make the algorithm runs more randomly but by extension, we could devalue the final result as we could make a choice that isn't really efficient at this time of the computation.

In order to try to find the optimal value, we need to choose how many iterations per run are processed. Most commonly this number is $t = 10$, but since we saw that most files were running fast enough, we set it to 30, this allowed us to prevent some cases where the algorithm didn't found the optimal value because of a lack of luck basically.

For the multidimensional constraints, as we are built around the greedy approach there isn't anything different from this methodology.

4.7 Greedy

For the greedy approach, even if we had to build three different versions based on the selector, we ended up building the same sorting method. So we have those three versions :

- Greedy by value : sorting using the value of objects.
- Greedy by weight : sorting using the wieght of objects.
- Greedy : sorting using the ratio of object value divide by object weighth.

For sorting all objects, we choose to use the TIMSORT sorting methods. It consists of mixing the insertion sort and the merge sort to increase it's speed. We first need to divide the list of objects into smaller groups of size such that $size_sub_list \in G$ such that $G = \{all\ number\ based\ on\ a\ power\ of\ two\}$. We set the minimal value to 32.

For the Multidimensional constraints, there isn't a lot of change with 0/1 constraints. We only change the weight calculus to take all dimensions into account.

4.8 Personal approach

We needed to think of a personal approach that isn't one already asked. That's where the brute force comes handy, as we could already see from the tests used as implementation validation for all algorithms (to be certain they all worked as intended). We could already spot the weakness of brute force.

That's when Alexandre got an idea, he thoughts of an adaptation of the divide and conquer algorithm called `meet-in-the-middle`.

This methodology consists of splitting the input list of objects in two, and find all subsets for both half-list. Because the brute force algorithm was already built, we simply call it on both of those sub list, and then we choose the best object and the second sub list, in order to add them into the first until we can't anymore.

5 Tests

5.1 Test procedure

Every tests were made following this protocol : It was done on a laptop with those specifications :

- Processor AMD Ryzen 5 4600H
 - 6 cores and 2 sockets (12 virtual cores) running at 4Ghz
- architecture x86_64
- 8 GiB of RAM running at 3200MHz

Only the process `launcher.py` was launched during the test periods (at the exception of all daemon process and the terminal).

For the ant colony and GRASP, all values are a median of multiple tests (10 runs per Algo), to ensure that we approximate the best way the value as it includes some randomness.

We decided to split the algorithms that have similar functioning into 4 Categories in order to make the tests :

- Brute Force, Branch and Bound and Personnel approach
- Dynamic Programming and FPTAS
- Ant Colony and GRASP
- Greedies algorithms

Once we had determined the best of each category, we have been able to test around the 4 remaining algorithms to determine the best one.

For the Solution accuracy, we launched each algorithms 10 times on the test files and took the median of all the solutions that had been found.

5.2 Test Results on 0/1 Problems

5.2.1 Time Complexity

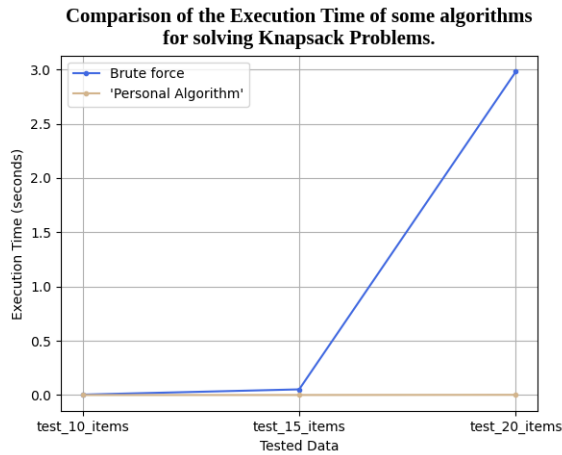


Figure 1: Comparison between Brute Force and Personal Algorithm

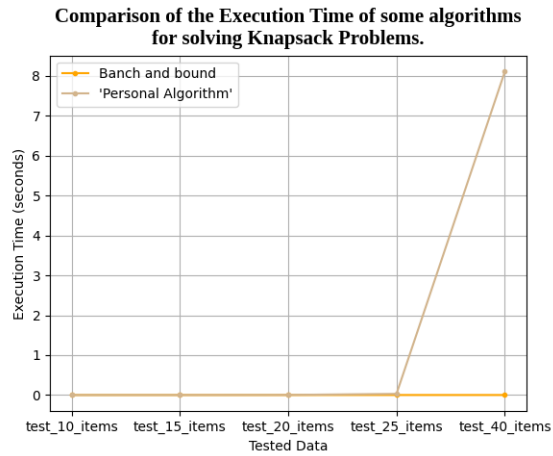


Figure 2: Comparison between Personal Algorithm and Branch and Bound

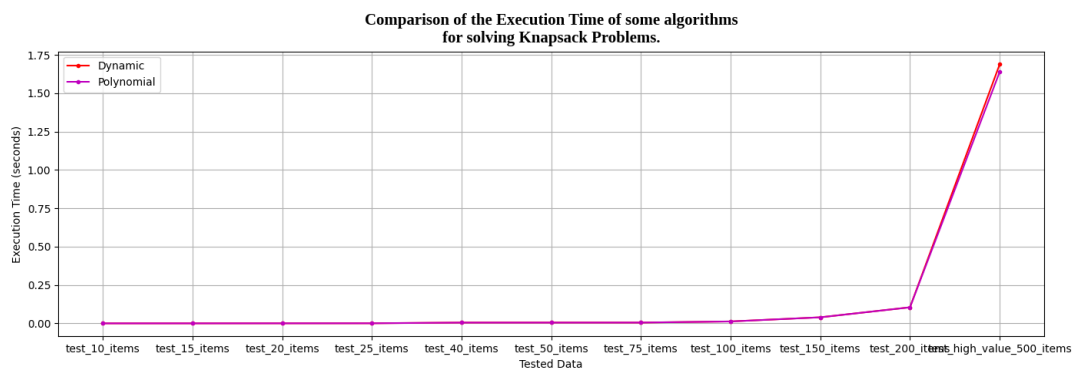


Figure 3: Comparison between Dynamic Programming and FPTAS

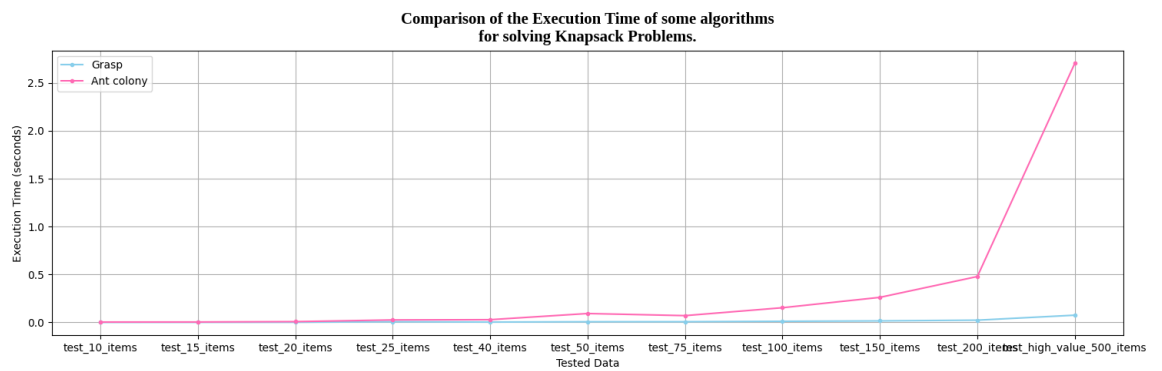


Figure 4: Comparison between Ant Colony and GRASP

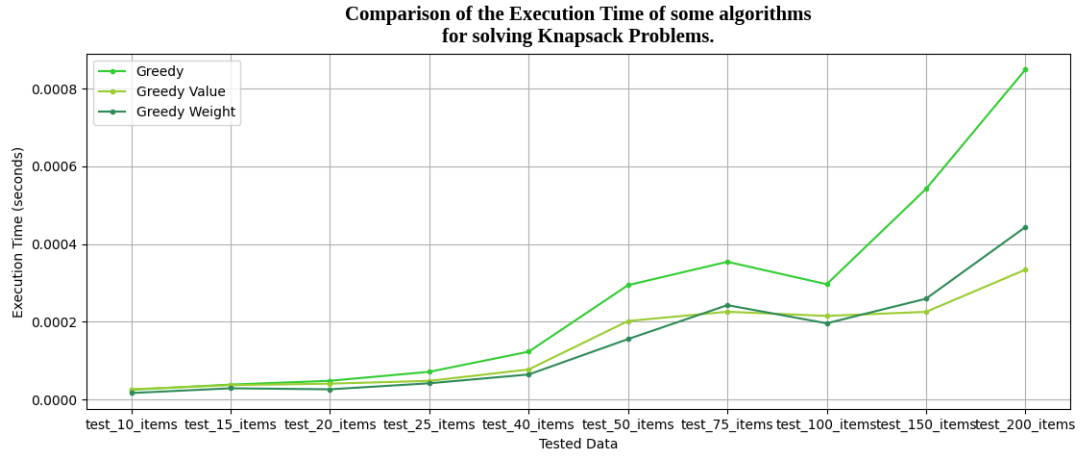


Figure 5: Comparison between Greedies algorithms

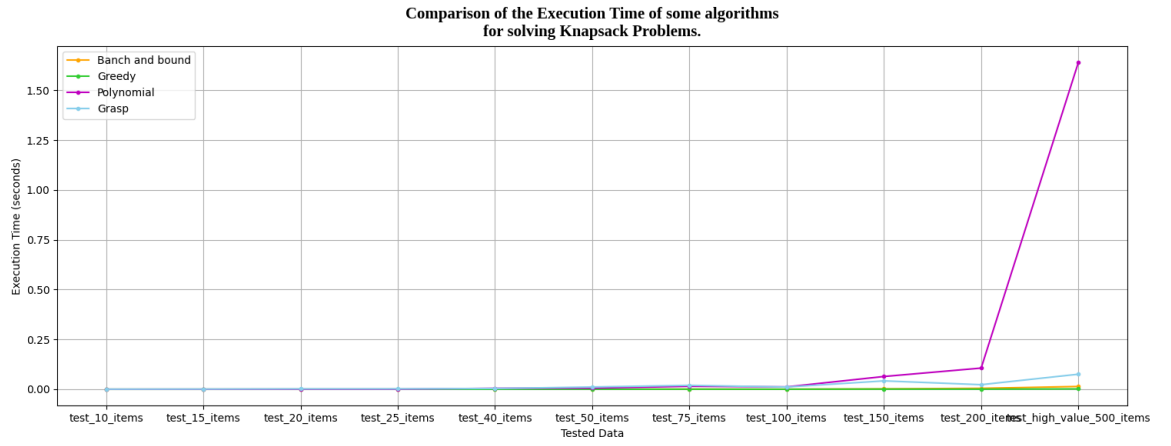


Figure 6: Comparison between the Best One of Each Category

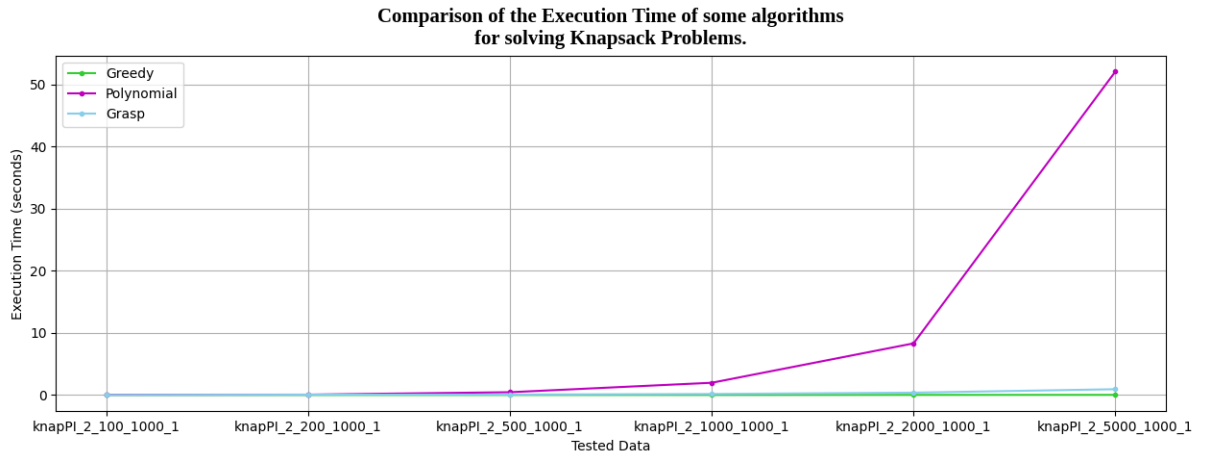


Figure 7: Comparison between GRASP, all greedy and polynomial

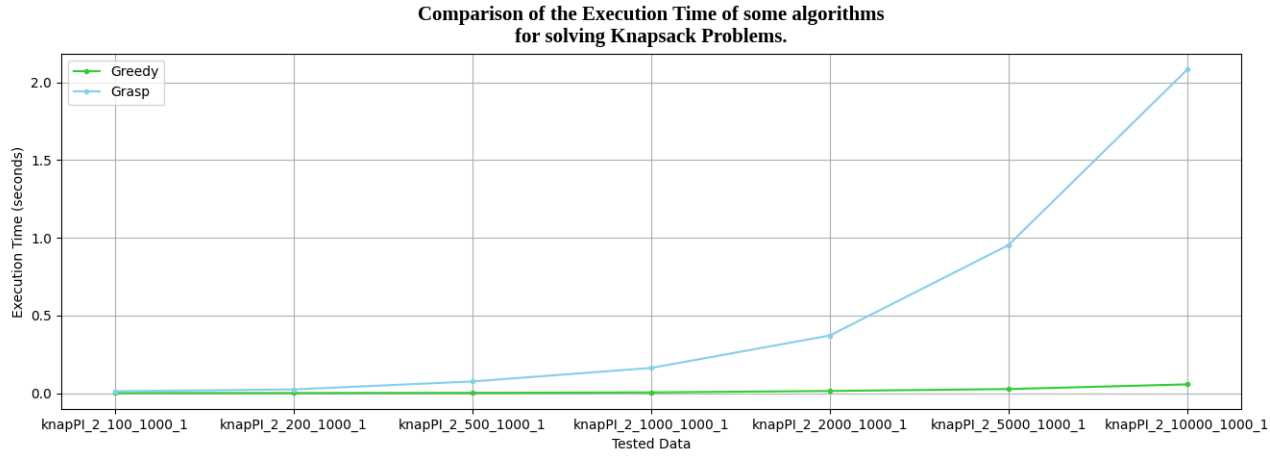


Figure 8: Comparison of the two best at very high numbers.

5.2.2 Solution Accuracy

file	brute_force	personal	branch_and_bound	ant_colony	grasp	dynamic	greedy_value	greedy_weight	greedy	polynomial
test_10_items	47	34	47	47	47	47	39	38	40	47
test_15_items	66	36	66	66	66	66	25	51	58	65
test_20_items	173	162	173	173	173	173	111	162	173	173
test_25_items	X	95	100	100	100	100	40	82	80	100
test_40_items	X	818	1000	1000	1000	1000	697	797	1000	1000
test_50_items	X	X	1229	1229	1229	1229	386	1116	1229	1229
test_75_items	X	X	1394	1386	1394	1394	296	1160	1307	1394
test_100_items	X	X	3898	3898	3898	3898	797	2564	3898	3898
test_150_items	X	X	10053	8899	10053	10053	899	8401	10053	10053
test_200_items	X	X	12188	10285	12188	12188	1977	10139	11835	12188
test_500_items	X	X	514500	486610	514500	514741	169571	492267	503414	514741
KnapPI_2_2000_1000_1	X	X	X	13824	18043	18051	10734	11972	17834	18051

Figure 9: Table representing the Solution found by each algorithm.

5.3 Test Results on Multidimensional Problems

5.3.1 Time Complexity

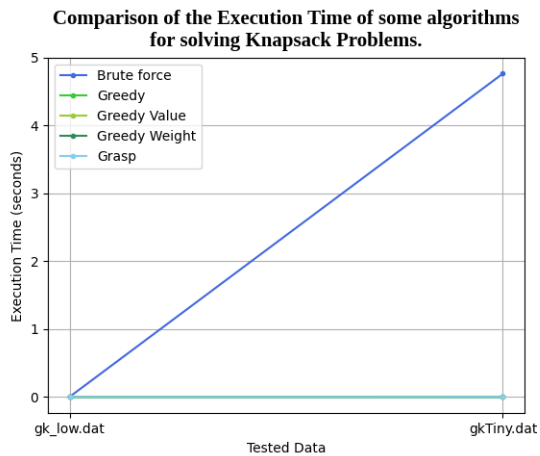


Figure 10: Comparison between GRASP, Brute Force and all Greedy

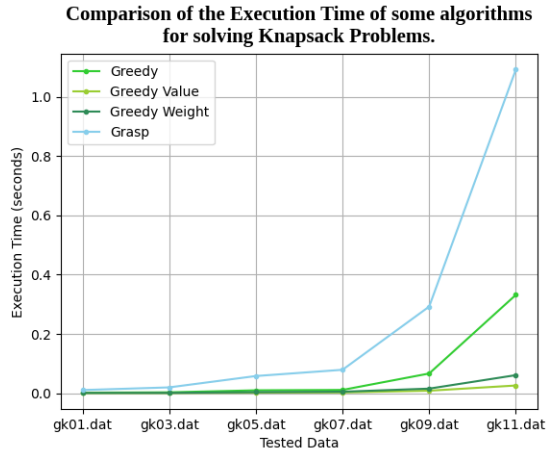


Figure 11: Comparison between GRASP and all Greedy

5.3.2 Accuracy

file	brute_force	grasp	greedy_value	greedy_weight	greedy
gk_low	160	160	160	116	160
gkTiny	0	0	0	0	0
gk01	X	3932	3863	3443	3860
gk03	X	5328	5192	4805	5246
gk05	X	7567	7509	6517	7566
gk07	X	20247	20241	17681	20220
gk09	X	62744	63010	55604	62736
gk11	X	100684	101154	83305	100681

Figure 12: Table representing the Solution found for each algorithm.

5.4 Results Interpretation

5.4.1 0/1 Problems

From all of our tests, we can see that brute force and our personal approach aren't very effective, even if brute force finds everytime the optimal value, it can't be processed beyond 25 objects (with 8Gib of RAM). Our personal value that tried to reduce the space complexity of brute force algorithm worked as we can reach almost twice as much objects processed, but in the computation we lost the ability to always find the optimal value and so make it a not so worthy algorithm.

In opposition we have the branch and bound algorithm that manages to find the optimal value, but we reach a limitation. This limitation is the allowed number of recursive call managed by the processor, and the algorithm can't take more than a 1000 objects before reaching that limitation. Now we only have to compare all greedys, the dynamic and fptas, as well as GRASP and ant colony.

Between all Greedy we spot that in general, the real greedy tends to work better at the exception of some files that are better suited for greedy by weight. Greedy by value is without a doubt the worst one as he doesn't even come close to the optimal value. But even if greedy is fast, it isn't close to the accuracy of all lasting algorthihms.

Ant colony seems a bit too random to be used realisticiy in comparison the GRASP, dynamic and polynomial algorithms. With that said it is still capable of putting out a fight against all other algorithms.

From the table of values found, we could think that GRASP algorithm isn't equivalent to the dynamic and polynomial algorithms. But even if GRASP could sometimes not found the optimal

value (because of it's randomness), it is capable of processing files beyond the 10000 objects in a reasonable run time (just a couple of seconds) whereas dynamic and polynomial algorithms aren't capable to process huge datasets (More than 2000 objects).

5.5 Multidimensional Problems

Our multidimensional tests are a little less important, as we didn't have as much algorithm to test the datasets on.

So, from our results, brute force algorithm isn't realistically usable as he can't process more than 20 objects. So let's look at the greedys and GRASP.

Without surprise from the already seen 0/1 results, the greedys work fast but aren't that efficient whereas GRASP is capable of managing it's run time while being able to find the optimal value. Speaking of optimal value, we were surprised to see to see that the `gk01.dat` file had an optimal solution smaller than our found GRASP value. We firstly though that our implementation wasn't working fine, and tried to correct it. After this we added a checker program (style available in the comments of `Grasp.py`), that checks if the final knapsack found didn't broke any dimension. To our surprise, it worked just fine, in confirmation of the checker program. So now we tend to think that maybe the datasets have an error on the optimal value of this sets. To confirm this idea even more, Our GRASP algorithm finds all the other optimal value from all other files.

6 Workload

Table 3: Table representing the Workload in percentage for each group member.

Name	Percentage
Alexandre MARINE	35%
Pierre BONNEFOY	20%
Ian RIVIERE	15%
Ophélie MARECHAL	15%
Aloys LANA	15%

7 Conclusion

Now that we studied all of our algorithms, we can try to evaluate them. From our tests results, we can already say that brute force, personal approach, greedy by value and branch and bound aren't really effective outside of our test conditions, as they all have some big drawbacks from within their logic.

As for greedy, greedy by weight and ant colony, they do not have enough reliability considering the value ratio on the optimal value awaited. For dynamic programming and polynomial approach, they aren't capable of processing huge datasets, and this is a very big drawback when we look at the modern problems and much data we have to process.

So we end up with the most effective algorithm (from our tests results), being GRASP algorithm. Even if sometimes it may not found the optimal value because of its randomness. It always gives a very good approximation at the very least and can do it on very big datasets in more than acceptable process times (order of the seconds for 10000 objects).

A Annexes

A.1 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? No, we think we have made enough rich commentaries.
13. Did you provide the additional planning and the final planning in the report and discuss organization aspects in your work? Yes
14. Did you provide the workload percentage between the members of the group in the report? Yes
15. Did you send the work in time? Yes

A.2 Algorithms

List of Algorithms

1	Branch_and_Bound	16
2	Branch_and_Bound_Recursive	16
3	Bound	17
4	GenerateAllPossibilities	18
5	Brute Force for 0/1 Knapsack Problems	18
6	Brute Force for Multidimensional Knapsack Problems	19
7	Greedy	20
8	Greedy (follow up)	21
9	GRASP	22
10	Ant Colony	23
11	Dynamic programming	24
12	Fully polynomial time approximation scheme	25

Algorithm 1: Branch_and_Bound

Data: *listItems* : Item list, *nbItems* : Number of items, *w* : Capacity of the knapsack
Result: *finalItems* : Solution item list, *finalValue* : Solution value
Global *NB_ITEMS*, *W*, *finalWeight*, *finalValue*, *finalSolution*, *tempSolution*
/* Initialisation */
NB_ITEMS \leftarrow *nbItems* ; *W* \leftarrow *w*
sort *listItems* **by value/weight**
finalWeight \leftarrow 0 ; *finalValue* \leftarrow 0 ; *finalSolution* \leftarrow []
currentWeight \leftarrow 0 ; *currentValue* \leftarrow 0 ; *tempSolution* \leftarrow []
while *len(tempSolution)* < *NB_ITEMS* **do**
 tempSolution.append(0)

/* Call of the recursive function */
Branch_and_Bound_Recursive(*listItems*, *currentValue*, *currentWeight*, 0)

/* Returned data */
fill *finalItems* list with the items selected in *finalSolution* (where
 finalSolution[*i*] == 1)
return *finalItems*, *finalValue*

Algorithm 2: Branch_and_Bound_Recursive

Data: *listItems* : Item list, *currentWeight* : Weight of the current solution,
 currentValue : Value of the current solution, *i* : index of the current item
Result: fill *finalWeight*, *finalValue* and *finalSolution*
Global *NB_ITEMS*, *W*, *finalWeight*, *finalValue*, *finalSolution*, *tempSolution*
/* If the current item *i* can be taken */
if *currentWeight* + *listItems*[*i*][1] \leq *W* **then**
 tempSolution[*i*] \leftarrow 1
 if *i* < *NB_ITEMS* - 1 **then**
 Branch_and_Bound_Recursive(*listItems*, *currentValue* +
 listItems[*i*][0], *currentWeight* + *listItems*[*i*][1], *i* + 1)
 if (*i* = *NB_ITEMS* - 1) **and** (*currentValue* + *listItems*[*i*][0] > *finalValue*) **then**
 finalValue \leftarrow *currentValue* + *listItems*[*i*][0]
 finalWeight \leftarrow *currentWeight* + *listItems*[*i*][1]
 finalSolution \leftarrow *tempSolution*

/* If the following items can lead to a better solution */
if *Bound*(*listItems*, *currentValue*, *currentWeight*, *i*) \geq *finalValue* **then**
 tempSolution[*i*] \leftarrow 0
 if *i* < *NB_ITEMS* - 1 **then**
 Branch_and_Bound_Recursive(*listItems*, *currentValue*, *currentWeight*, *i* +
 1)
 if (*i* = *NB_ITEMS* - 1) **and** (*currentValue* > *finalValue*) **then**
 finalValue \leftarrow *currentValue* ; *finalWeight* \leftarrow *currentWeight*
 finalSolution \leftarrow *tempSolution*

Algorithm 3: Bound

Data: *listItems* : Item list, *currentWeight* : Weight of the current solution,
 currentValue : Value of the current solution, *i* : index of the current item
Result: *v* : Maximum value reachable using the items following the current item *i*
Global *NB_ITEMS*, *W*
v \leftarrow *currentValue*
w \leftarrow *currentWeight*
for *j* \leftarrow *i* + 1 **to** *NB_ITEMS* **do**
 if *w* + *listItems*[*j*][1] \leq *W* **then**
 w \leftarrow *w* + *listItems*[*j*][1]
 v \leftarrow *v* + *listItems*[*j*][0]
return *v*

Algorithm 4: GenerateAllPossibilities

Data: n : Number of bits
Result: List of every sequence of n bits possible.

```
 $l \leftarrow [[0], [1]];$   
 $start \leftarrow 0;$   
for  $i \leftarrow 1$  to  $n$  do  
   $tmp \leftarrow len(l);$   
  for  $element$  in  $l[start : ]$  do  
     $l \leftarrow l.append([0] + element);$   
     $l \leftarrow l.append([1] + element);$   
   $start \leftarrow tmp;$   
return  $l[start : ]$ 
```

Algorithm 5: Brute Force for 0/1 Knapsack Problems

Data: $maxWeight$: Capacity of the Knapsack, $objectList$: List of object of the problem.
Result: Final Knapsack, Final Value

```
 $allPossibilities \leftarrow GenerateAllPossibilities(len(objectList));$   
 $currentBest \leftarrow (0, -1);$   
for  $currentPossibility, element$  in enumerate  $allPossibilities$  do  
   $currentWeight \leftarrow 0;$   
   $currentValue \leftarrow 0;$   
  for  $index, bit$  in enumerate  $element$  do  
    if  $bit == 1$  then  
       $currentWeight \leftarrow currentWeight + objectList[index][1];$   
      if  $currentWeight \leq maxWeight$  then  
         $currentValue \leftarrow currentValue + objectList[index][0];$   
      else  
        break  
    if  $currentValue > currentBest[1]$  then  
       $currentBest \leftarrow (currentPossibility, currentValue);$   
  
  /* Reconstitution Of Knapsack for Displaying Purposes. */  
   $finalKnapsack \leftarrow [];$   
  for  $i, obj$  in enumerate  $allPossibilities[currentBest[0]]$  do  
    if  $obj == 1$  then  
       $finalKnapsack \leftarrow finalKnapsack.append([objectList[i]]);$   
  
  return  $finalKnapsack, currentBest[1]$ 
```

Algorithm 6: Brute Force for Multidimensional Knapsack Problems

Data: *ksc* : List of the capacity of each dimension of the Knapsack, *objectList* : List of object of the problem.

Result: Final Knapsack, Final Value

allPossibilities \leftarrow *GenerateAllPossibilities*(*len(objectList)*);

currentBest \leftarrow (0, -1);

nbDimension \leftarrow *len(ksc)*;

for *currentPossibility, element* **in** *enumerate allPossibilities* **do**

currentWeigth \leftarrow *initializeWith0*(*nbDimension*);

currentValue \leftarrow 0;

for *index, bit* **in** *enumerate element* **do**

if *bit* == 1 **then**

for *dimension* \leftarrow 0 **to** *nbDimension* **do**

currentWeigth[*dimension*] \leftarrow

currentWeigth[*dimension*] + *objectList*[*index*][1][*dimension*];

if *not*(*currentWeigth* <= *maxWeigth*) **then**

break

else

currentValue \leftarrow *currentValue* + *objectList*[*index*][0];

continue

break

if *currentValue* > *currentBest*[1] **then**

currentBest \leftarrow (*currentPossibility*, *currentValue*);

/* Reconstitution Of Knapsack for Displaying Purposes. */

finalKnapsack \leftarrow [];

for *i, obj* **in** *enumerate allPossibilities*[*currentBest*[0]] **do**

if *obj* == 1 **then**

finalKnapsack \leftarrow *finalKnapsack.append*(*objectList*[*i*]);

return *finalKnapsack*, *currentBest*[1]

Algorithm 7: Greedy

Data: *list_objects*, *wmax*, a list of tuple object with their profit and weight and the maximum capacity of the knapsack.

Function Main(*list_objects*, *wmax*):

```
    start_timer
    list_temp ← list_objects[:]
    cw ← wmax
    final_list_objects ← []
    final_value ← 0
    timsort(list_temp)
    for i to len(list_temp) do
        if cw − list_temp[i][1] > 0 then
            cw − = list_temp[i][0]
            fw + = list_temp[i][0]
            final_knapsack.append(list_temp[i])
        else
            break
    end_timer
```

Result: *time_taken*, *final_knapsack*, *final_value*

Function Timsort(*list_objects*):

```
    n ← len(list_objects)
    minrun ← find_minrun(n)
    for start ← minrun to n do
        end ← min(start + (minrun − 1), n − 1)
        insertion_sort(list_objects, start, end)
    size ← minrun
    while size < n do
        for left ← 2 * size to n do
            mid ← min(n − 1, (left + size − 1))
            right ← min((left + 2 * size − 1), n − 1)
            if mid < right then
                merge(list_objects, left, mid, right)
        size ← 2 * size
```

Result: *sorted_list*, the input list sorted.

Function Find_minrun(*n*):

```
    r ← 0
    /* MINIMUM is set globally to 32 */
    while n >= MINIMUM do
        r ← n|1
        /* Binary OR comparison between the bits value of n and 1 */

        n ←>> 1
        /* Binary shifting of the bits value of 1. Corresponds to dividing
           n by 2**1 */
```

Result: *minimal_run_size*, the runnable size that will be used to split the lists.
Will always be a power of 2.

Algorithm 8: Greedy (follow up)

Function Insertion_sort(*list_objects*, *left*, *right*):

```
    for i from left + 1 to right + 1 do
        j = i
        while j > left and
            list_objects[j][0]/list_objects[j][1] > list_objects[j - 1][0]/list_objects[j - 1][1]
        do
            list_objects[j], list_objects[j - 1] ← list_objects[j - 1], list_objects[j]
            j --
```

Result: *sorted_list*, using insertion methods on the left and right parts of input.

Function Merge(*list_objects*, *left*, *mid*, *right*):

```
    len1 ← mid - left + 1
    len2 ← right - mid
    left_part, right_part ← []
    for i ← 1 to len1 do
        left_part.append(list_objects[left + i])
    for i ← 1 to len2 do
        right_part.append(list_objects[mid + 1 + i])
    i, j ← 0
    k ← left
    while i < len1 and j < len2 do
        if left_part[i][0]/left_part[i][1] < right_part[j][0]/right_part[j][1] then
            list_objects[k] ← right_part[j]
            j ++
        else
            list_objects[k] ← left_part[i]
            i ++
        k ++
    while i < len1 do
        list_objects[k] ← left_part[i]
        k ++
        i ++
    while j < len2 do
        list_objects[k] ← right_part[j]
        k ++
        j ++
```

Result: *merged_list*, a sorted version of *list_objects* based on a merge version of it's left and right parts.

Algorithm 9: GRASP

Data: *list_objects*, *nbI*, *wmax*, a list of tuple object with their profit and weight, the number of Iterations and the maximum capacity of the knapsack.

Function Main(*list_objects*,*nbI*,*wmax*):

```
    start_timer
    final_list, final_value ← grasp(list_objects, nbI, wmax)
    end_timer
```

Result: *time_taken*, *final_knapsack*, *final_value*

Function GRASP(*list_objects*,*nbI*,*wmax*):

```
    best ← 0
    list_temp ← []
    solution ← []
    v ← 0
    for i to nbI do
        list_temp, v ← greedy_randomised_construction(list_objects, wmax)
        if v > best then
            best ← v
            solution ← list_temp[:]
```

Result: *solution*, *best*

Function Greedy_randomised_construction(*list_objects*, *wmax*):

```
    list_temp ← list_objects[:]
    cw ← wmax
    final_list ← []
    final_value ← 0
    timsort(list_temp)
    while len(list_temp) > 0 do
        current_candidates ← []
        for i to 3 do
            if i < len(list_temp) then
                current_candidates.append(list_temp[i])
        chosen_object ← random.choice(current_candidates)
        if chosen_object[1] ≤ cw then
            final_list.append(chosen_object)
            cw − = chosen_object[1]
            final_value + = chosen_object[0]
            list_temp.pop(list_temp.index(chosen_object[0]))
```

Result: *final_list*, *final_value*

Algorithm 10: Ant Colony

Data: *list_objects*, *nb_ant*, *nb_objects*, *wmax*
Result: *best_solution*['objects'], *best_solution*['value']
init_phero(*phero*);
best_solution \leftarrow {'num_object' = [], 'objects' = [], 'value' = -1, 'weight' = -1};
for *ant* = 1 **to** *nb_ant* **do**
 current_solution \leftarrow {'num_object' = [], 'objects' = [], 'value' = -1, 'weight' = -1};
 init_candidates(*candidates*, *list_objects*);
 check \leftarrow 0;
 current \leftarrow 0;
 set_probabilities(*probabilities*, *phero*, *list_objects*, *candidates*, *current*);
 x \leftarrow random between 0 and 1;
 foreach *i* in *probabilities* **do**
 if *x* \in *i* **and**
 current_solution['weight'] + *list_objects*[*i*['num_object']]['weight'] \leq *wmax*
 then
 remove *i*['num_object'] of *candidates*;
 remove *i*['num_object'] of *probabilities*;
 add *list_objects*[*i*['num_object']] to *current_solution*;
 current \leftarrow *i*['num_object'];
 check \leftarrow *check* + 1;
 while *check* \neq *nb_objects* **do**
 et_probabilities(*probabilities*, *phero*, *list_objects*, *candidates*, *current*);
 x \leftarrow random between 0 and 1;
 foreach *i* in *probabilities* **do**
 if *x* \in *i* **and**
 current_solution['weight'] + *list_objects*[*i*['num_object']]['weight'] \leq *wmax* **then**
 remove *i*['num_object'] of *candidates*;
 remove *i*['num_object'] of *probabilities*;
 add *list_objects*[*i*['num_object']] to *current_solution*;
 current \leftarrow *i*['num_object'];
 check \leftarrow *check* + 1;
 else
 if *current_solution*['weight'] + *list_objects*[*i*['num_object']]['weight'] \leq *wmax* **then**
 remove *i*['num_object'] of *candidates*;
 remove *i*['num_object'] of *probabilities*;
 check \leftarrow *check* + 1;
 if *current_solution*['value'] > *best_solution*['value'] **then**
 best_solution \leftarrow *current_solution* ;
 update_phero(*phero*);
 set_phero(*phero*, *best_solution*, *current_solution*);

Algorithm 11: Dynamic programming

Data: *list_objects*, *wmax*, a list of tuple object with their profit and weight, the maximum capacity of the knapsack

Function *Dynamic programming*(*list_objects*, *wmax*):

```
    start_timer
    list_temp ← list_objects
    final_list ← []
    final_value ← 0
    /* Creating the matrix of 0 to n objects as rows and 0 to Knapsack max
       weight */
    matrix = [0] * (w + 1) * (len(list_temp) + 1)
    /* Filling the table */
    for i to len(list_temp) + 1 do
        for j to wmax + 1 do
            if i == 0 or j == 0 then
                matrix[i][j] ← 0
            else if list_temp[i - 1][1] <= j then
                matrix[i][j] ← max(list_temp[i - 1][0] + matrix[i - 1][j - list_temp[i - 1][1]], matrix[i - 1][j])
            else
                matrix[i][j] ← matrix[i - 1][j]
        end for
    end for
    final_value = matrix[len(list_temp)][wmax]
    checking_value ← final_value
    checking_weight ← wmax
    /* Getting all selected items */
    for i ← -1 from len(list_temp) to 0 do
        if checking_value <= 0 then
            break
        if checking_value == matrix[i - 1][checking_weight] then
            continue
        else
            final_list.append(list_temp[i - 1])
            checking_value − = list_temp[i - 1][0]
            checking_weight − = list_temp[i - 1][1]
        end if
    end for
    end_timer
```

Result: *final_list*, *final_value*

Algorithm 12: Fully polynomial time approximation scheme

Data: ϵ , w_{\max} , $list_objects$

Function $F_{ptas}(\epsilon, w_{\max}, list_objects)$:

```
    start_timer
    final_value  $\leftarrow$  0
    final_list  $\leftarrow$  []
     $P \leftarrow list\_objects[0][0]$ 
    for  $i$  from 1 to  $len(list\_objects)$  do
        if  $list\_objects[i][0] > P$  then
             $P \leftarrow list\_objects[i][0]$ 
     $K \leftarrow (\epsilon * P) / len(list\_objects)$ 
     $list\_temp \leftarrow [(int(a[0]/K), a[1]) \text{ for } a \in list\_objects]$ 
    /* It's a slightly modified version of Dynamic programming returning
       the list of index instead of the final_list */
     $list\_index \leftarrow dynamic(list\_temp, w_{\max})$ 
    for  $i \in list\_index$  do
        final_sac.append( $list\_objects[i]$ )
        final_value +  $= list\_objects[i][0]$ 
    end_timer
```

Result: $final_list, final_value$
