

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	3
3.1	Tasks Repartition	3
3.2	Planification	4
4	Implementation Choices	4
4.1	Ant Colony	4
4.2	Branch and Bound	4
4.3	Brute Force	5
4.4	Dynamic Programming	5
4.5	Fully polynomial time approximation scheme	5
4.6	GRASP	5
4.7	Greedy	6
4.8	Personal approach	6
5	Tests	7
5.1	Test procedure	7
5.2	Test Results	7
5.3	Results Interpretation	7
6	Workload	7
7	Conclusion	7
A	Annexes	8
A.1	Algorithms	9
A.2	Checklist	13

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](#)

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
FPETAS 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

5.2 Test Results

5.3 Results Interpretation

6 Workload

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

Name	Percentage
Alexandre MARINE	35%
Pierre BON- NEFOY	20%
Ian RIV- IERE	15%
Ophélie MARECHAL	15%
Aloys LANA	15%

7 Conclusion

A Annexes

List of Algorithms

1	Branch_and_Bound	9
2	Branch_and_Bound_Recursive	10
3	Bound	10
4	GenerateAllPossibilities	11
5	Brute Force for 0/1 Knapsack Problems	11
6	Brute Force for Multidimensional Knapsack Problems	12
7	Greedy	14
8	Greedy (follow up)	15
9	GRASP	16
10	Ant Colony	17

A.1 Algorithms

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] ≤ W then
    tempSolution[i] ← 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 ← currentValue + listItems[i][0]
        finalWeight ← currentWeight + listItems[i][1]
        finalSolution ← tempSolution

/* If the following items can lead to a better solution */
if Bound(listItems, currentValue, currentWeight, i) ≥ finalValue then
    tempSolution[i] ← 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 ← currentValue ; finalWeight ← currentWeight
        finalSolution ← 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 ← currentValue
w ← currentWeight
for j ← i + 1 to NB_ITEMS do
    if w + listItems[j][1] ≤ W then
        w ← w + listItems[j][1]
        v ← 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**

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

currentValue \leftarrow 0;

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

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

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

currentWeigh[*dimension*] \leftarrow

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

if *not*(*currentWeigh* <= *maxWeigh*) **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]

A.2 Checklist

1. Did you proofread your report?
2. Did you present the global objective of your work?
3. Did you present the principles of all the methods/algorithms used in your project?
4. Did you cite correctly the references to the methods/algorithms that are not from your own?
5. Did you include all the details of your experimental setup to reproduce the experimental results, and explain the choice of the parameters taken?
6. Did you provide curves, numerical results and error bars when results are run multiple times?
7. Did you comment and interpret the different results presented?
8. Did you include all the data, code, installation and running instructions needed to reproduce the results?
9. Did you engineer the code of all the programs in a unified way to facilitate the addition of new methods/techniques and debugging?
10. Did you make sure that the results different experiments and programs are comparable?
11. Did you sufficiently comment your code?
12. Did you add a thorough documentation on the code provided?
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?
15. Did you send the work in time?

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 */
    end
```

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  $\leftarrow$  grasp(list_objects, nbI, wmax)  
    end_timer
```

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

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

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

Result: *solution*, *best*

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

```
    list_temp  $\leftarrow$  list_objects[:]  
    cw  $\leftarrow$  wmax  
    final_list  $\leftarrow$  []  
    final_value  $\leftarrow$  0  
    timsort(list_temp)  
    while len(list_temp) > 0 do  
        current_candidates  $\leftarrow$  []  
        for i to 3 do  
            if i < len(list_temp) then  
                current_candidates.append(list_temp[i])  
        chosen_object  $\leftarrow$  random.choice(current_candidates)  
        if chosen_object[1] <= cw then  
            final_list.append(chosen_object)  
            cw = cw - chosen_object[1]  
            final_value = 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*);
