John-Michael JENY JEYARAJ

Rapport Conception D'algorithme

1. Un fichier .csv est un fichier où les valeurs sont séparées par des virgules. Après avoir compilé le programme c'est évident que le programme à créer un fichier Instances.csv, en format d'un tableau, avec la capacité du sac suivi par le nombre d'objet et suivi par des couple de valeur et poids pour chaque objet. Chaque valeur est aléatoire et séparée par une virgule et on passe à la prochaine valeur avec un token.

```
30,5,15,8,4,9,7,1,1,5,10,5,2
42,5,14,8,1,6,2,9,10,7,9,5,5
35,5,11,1,9,3,1,4,1,5,3,8,8
41,5,10,6,9,5,7,8,10,7,8,5,1
47,5,18,5,4,1,4,4,4,10,9,4,8
41,5,10,7,4,8,5,9,3,5,6,6,4
42,5,20,6,5,6,1,2,9,1,2,7,9
29,5,16,4,4,10,9,8,5,3,7,2,1
41,5,15,10,3,5,4,9,9,7,5,5,4
35,5,16,5,8,9,4,5,1,2,3,1,10
31,6,16,10,9,3,5,4,6,7,5,2,8,9,8
45,6,14,5,2,4,6,1,6,2,9,3,5,1,9
46,6,16,9,2,7,10,4,3,10,3,5,3,4,4
32,6,12,10,9,4,8,2,6,2,5,10,8,6,10
49,6,10,2,9,10,8,3,2,5,2,5,3,7,2
30,6,17,3,10,1,9,7,7,4,2,5,7,1,1
31,6,10,1,5,10,3,8,4,9,7,9,7,1,2
26,6,17,4,7,2,2,7,8,5,3,9,5,9,6
48,6,12,6,6,4,1,4,8,8,7,4,10,4,5
37,6,19,3,10,8,9,1,7,8,3,10,5,7,1
31,7,20,5,8,3,3,8,3,6,6,7,7,10,7,10,6
41,7,14,10,4,7,5,10,4,10,4,8,1,1,9,8,9
41,7,20,7,4,2,5,4,8,8,5,3,10,8,2,4,9
37,7,18,3,6,5,10,1,1,10,6,2,6,10,4,1,10
27,7,13,7,4,9,4,10,10,6,3,1,10,4,2,1,3
39,7,13,7,1,3,1,6,9,4,8,10,1,10,7,2,7
49,7,10,4,6,3,10,3,2,10,7,1,9,1,8,1,7
27,7,15,7,5,7,7,7,8,10,3,3,6,6,5,8,2
44,7,16,10,7,8,3,3,8,8,4,6,1,4,5,1,7
49,7,10,2,2,6,9,10,9,6,2,9,5,9,1,7,8
28,8,12,2,10,1,7,1,4,8,3,4,2,10,1,8,6,1,8
32,8,19,7,5,6,4,5,2,6,1,5,9,4,8,1,9,2,9
50,8,17,6,3,5,3,1,8,4,10,9,2,5,7,1,2,4,5
31,8,16,5,10,5,1,10,1,9,4,9,7,6,3,3,6,4,9
```

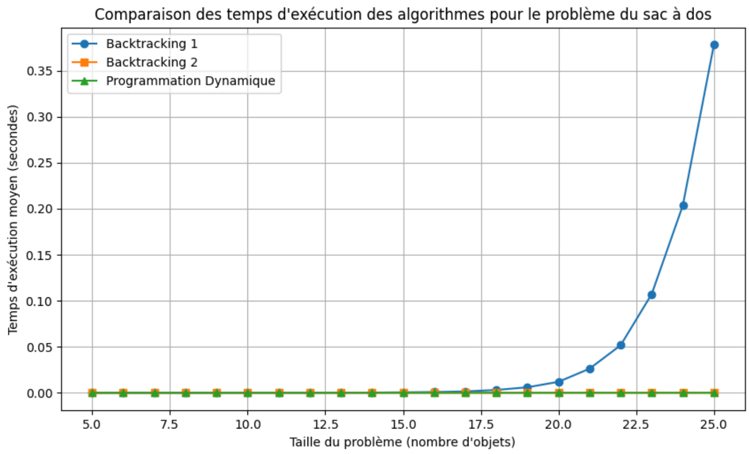
2. Les 2 fonctions, knapsackBT1 et knapsackBT2 ont la même complexité de temps O(2^n). Parce que l'algorithme décide s' il doit ajouter ou non chaque objet un par un.

   Dans knapsackBT1 on incrémente récursivement index jusqu'à index = n ou si on a atteint la capacité du sac. Puis si l'item à la position n-1 rentre dans le sac, on met à jour le bestValue avec la valeur de l'item à la position n-1 et on remet currentweight et currentvalue à 0. Puis on dépile et si les items à la position n-2 et n-1 rentre dans le sac, on compare leur valeur de (n-2) + (n-1) avec bestValue
   Puis on dépile et si les items à la position n-3, n-2 et n-1 rentrent dans le sac, on compare la valeur de (n-3) + (n-2) + (n-1) avec bestValue.

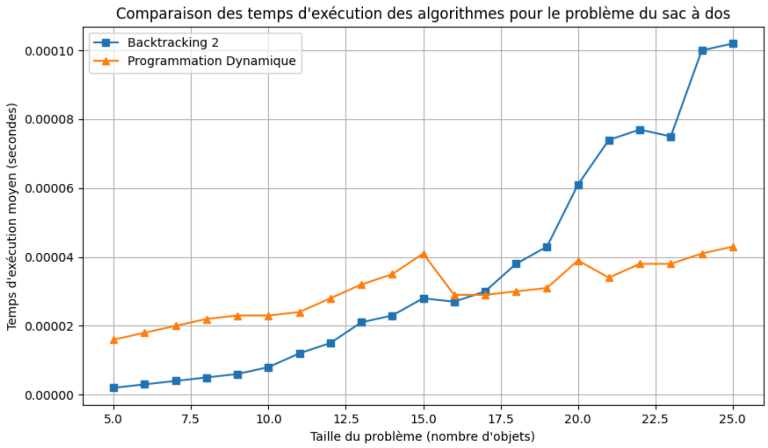
   Ici utilisé des variable globale est meilleur parce que sinon on empile et dépile les variables locale, qui prend beaucoup plus d'espace mémoire. Même sis d'habitude utilisé des variables global

3. Le programme devient trop lent quand le nombre d'objets est environ 24. Car il est inefficace est fait plusieurs boucles pour trouver une solution.

4. On intègre la borne supérieure dans la fonction borneSomme pour élaguer les calcules de l'arbre de recherche. La borne supérieure est calculée en prenant la valeur de l'objet qu'on considère et le poids de tous les objets après lui, et on regarde si il peut atteindre la solution optimale.



Il est beaucoup plus efficace que le backtracking sans l'élagage.

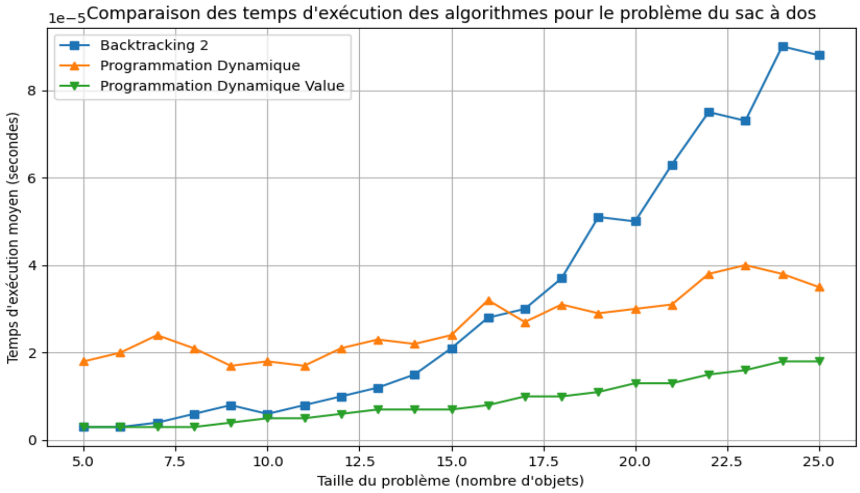


Il est même comparable au programmation dynamique avant n = 24.

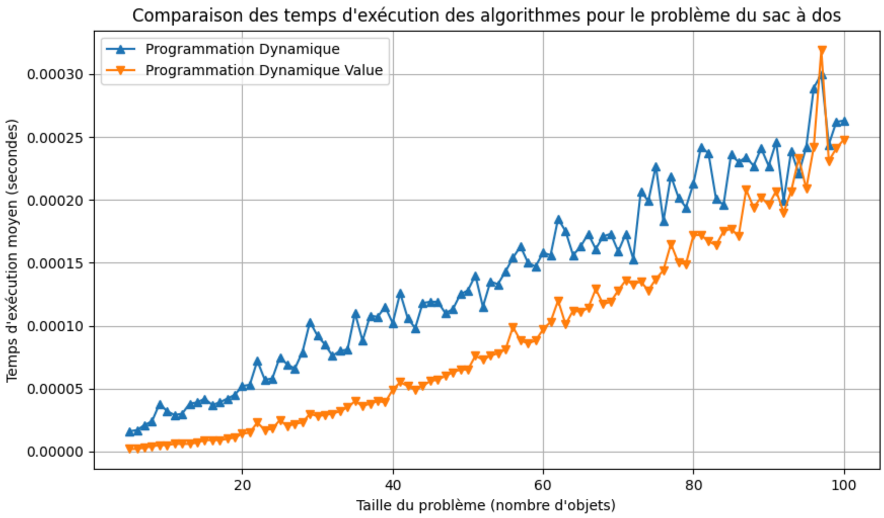
5. Pour montrer que le problème a la propriété de sous-structure optimale il suffit de montrer que pour n objets O_1, … O_n, si $O_{i,1}$, … $O_{i,k}$ est une solution optimale d'un knapsack avec un poid maximum de W, alors on a $O_{i,1}$, … $O_{i,k-1}$ une solution optimale d'un knapsack avec un poid maximum de W - $w_{i,k}$ où $w_{i,k}$ est le poid de $O_{i,k}$.
Par l'absurde, soit p la valeur de la solution optimale de O_1, … O_n. On suppose qu' il existe un sous-ensemble d'objets {O_1, …, O_n}\ {O_k} qui est la solution optimale de valeur p' > p - $p_{i,k}$ pour un knapsack avec un poid maximum W-$w_{i,k}$. Si on ajoute à cette solution l'objet $O_{i,k}$, (de poid $w_{i,k}$ et de valeur $p_{i,k}$) on aura une solution pour un knapsack de poid maximum W de valeur p'+$p_{i,k}$ >p. Ceci est absurde car la valeur de la solution optimale est p.

Définition récursive de dp[v]=min(dp[v],dp[v−wi]+pi)

6. On implémente une fonction knapsackDP_Value, qui utilise dp[v]. Dans le main on copie comment on calcule et affiche les valeurs de knapsackDP. Puis dans plot_temps.py on copie comment on illustre les données en une courbe. La complexité de l'algorithme est O(nV) car il dépend du nombre d’objets et de la somme de leur valeur.

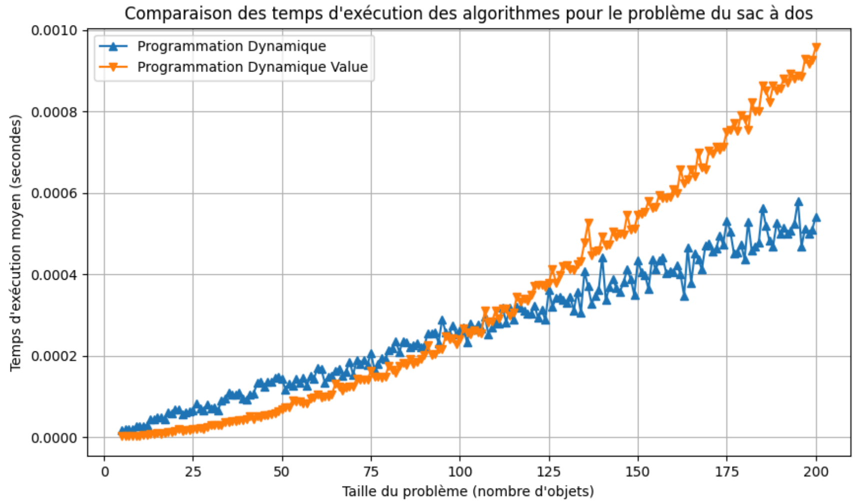


NUM_INSTANCES_PER_SIZE = 10
MAX_ITEMS = 25
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 200
MAX_CAPACITY = 200

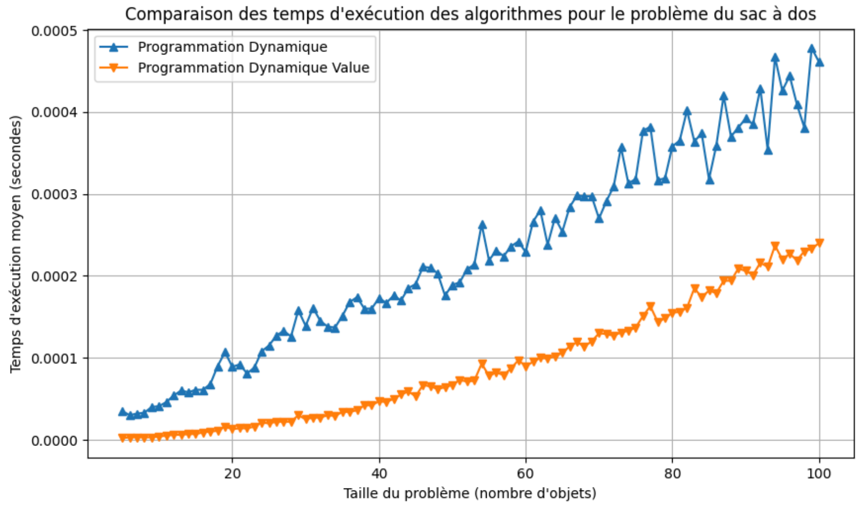


NUM_INSTANCES_PER_SIZE = 10
MAX_ITEMS = 100
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 200

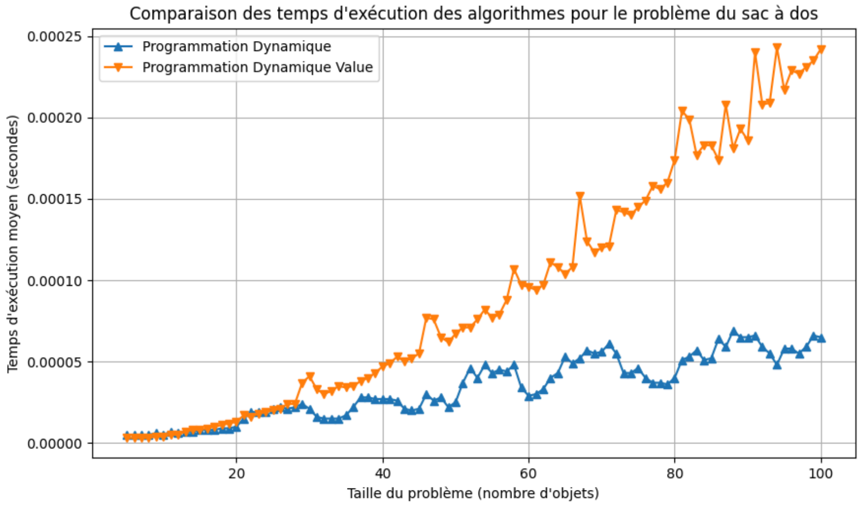
John-Michael JENY JEYARAJ



NUM_INSTANCES_PER_SIZE = 10
MAX_ITEMS = 200
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 200



NUM_INSTANCES_PER_SIZE = 10
MAX_ITEMS = 100
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 500



NUM_INSTANCES_PER_SIZE = 10
MAX_ITEMS = 100
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 50

On peut voir que la complexité de knapsackDP est meilleure quand la capacité du sac est petite, par exemple 50. Et la complexité de knapsackDP_Value est meilleure quand la capacité du sac est grande ou quand la somme des valeurs est petite.

7. La valeur qu'on stocke dans dp[i][w] est la valeur maximale des objets inclus dans le sac à dos quand on considère les i premiers objets et que le poids total du sac est w.

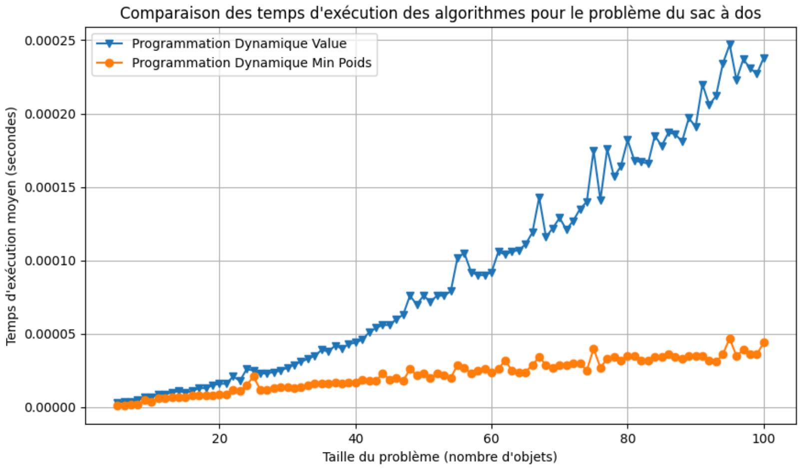
dp[i][w]=
dp[i−1][w] si w<items[i−1].weight
max(dp[i−1][w],dp[i−1][w−items[i−1].weight]+items[i−1].value) si w≥items[i−1].weight

On implémente l'algorithme avec une fonction et on change generateData.py pour créer un poid minimum aléatoire avec les modification:

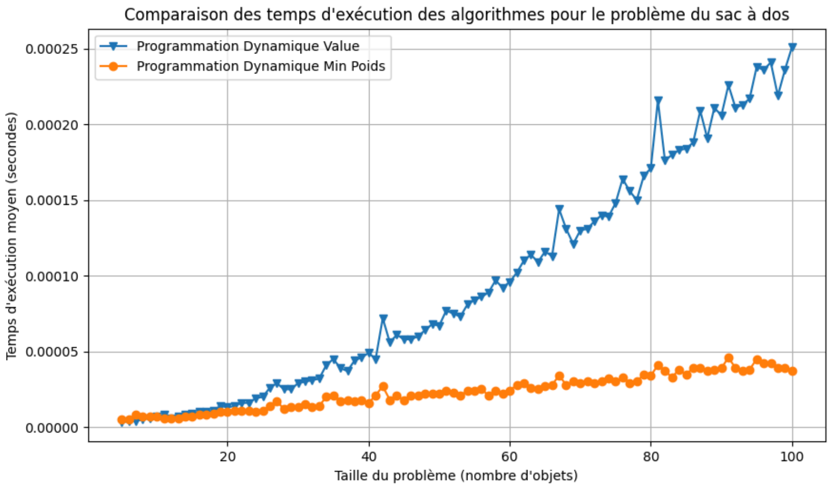
MIN_CAPACITY = 10

min_weight = random.randint(MIN_CAPACITY, 2* MIN_CAPACITY)

John-Michael JENY JEYARAJ



NUM_INSTANCES_PER_SIZE = 10
MAX_ITEMS = 100
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 50
MIN_WEIGHT = 10
NUM_INSTANCES_PER_SIZE = 10



MAX_ITEMS = 100
MIN_ITEMS = 5
ITEM_STEP = 1
MAX_WEIGHT = 10
MAX_VALUE = 10
MAX_CAPACITY = 50
MIN_WEIGHT = 80

La complexité de programmation dynamique avec un poid minimum est meilleure, sauf quand le le poid tout les objet est inférieur au poid minimum.

8. Pour implémenter l'algorithme gloutons, on trie avec le tri rapide dans l'ordre du plus grand rapport entre la valeur et le poid. Puis on ajoute les items au sac dans l'ordre du trie jusqu'à atteindre la capacité du sac ou si on à considérer tous les objets. L'exemple quand l'algorithme gloutons n'est pas optimal est si on un sac de capacité W et on a que 2 objet: un de valeur 1 et poid 1 et l'autre de valeur W-1 et poid W.

   Prenons W = 5, si le premier objet a un poids = 2 et une valeur = 2, et le deuxième a un poids = 5 et une valeur = 4. On trie le tableau dans l'ordre du rapport entre la valeur / le poid, et on prend le premier objet. Mais la solution optimale était de prendre le deuxième objet.

   Le premier objet à un meilleur rapport entre la valeur et le poids, mais on ne pourra pas ajouter le deuxième objet qui a une valeur beaucoup plus importante, malgré son rapport entre sa valeur et son poids. L'algorithme glouton est le plus rapide des algorithmes, mais il n'a souvent pas la solution optimale.
   Soit n le nombre d'objet, l'algorithme a une complexité spatiale O(n) linéaire et une complexité temporelle O(n*log(n)).