UNIVERSITÉ SORBONNE PARIS NORD

Institut Galilée

Rapport du projet de Conception d'Algorithmes

Résolution du Problème du Sac à dos 0/1

Formation : Licence 2 informatique
Matière : Conception d'algorithmes
Enseignant : Nadi Tomeh

Nom : Mehamli
Prénom : Samy
Numéro étudiant : 12213639

26 mai 2024

Table des matières

1 Exercice 1 : Reproduction

1.1 Question 1 : Format du fichier **instance.csv**

instance.csv est du format CSV(Comma-Separated Values), où les données sont organisées en lignes et colonnes, avec chaque colonne séparée par une virgule. Chaque ligne de ce fichier représente une instance de données, où le premier argument indique le poids maximal du sac à dos, le deuxième représente le nombre d'objets disponibles, et les arguments suivants sont les paires de poids et de valeurs de chaque objet, rangés comme suit : poids de l'objet 1, valeur de l'objet 1, poids de l'objet 2, valeur de l'objet 2, ..., poids de l'objet n, valeur de l'objet n. Chaque ligne est de cette forme :
W, n, w(1), v(1), w(2), v(2),..., w(n), v(n).

1.2 Question 2 : Comparaison des deux algorithmes de **Backtracking**

1.2.1 Backtracking 1 : avec variables globales

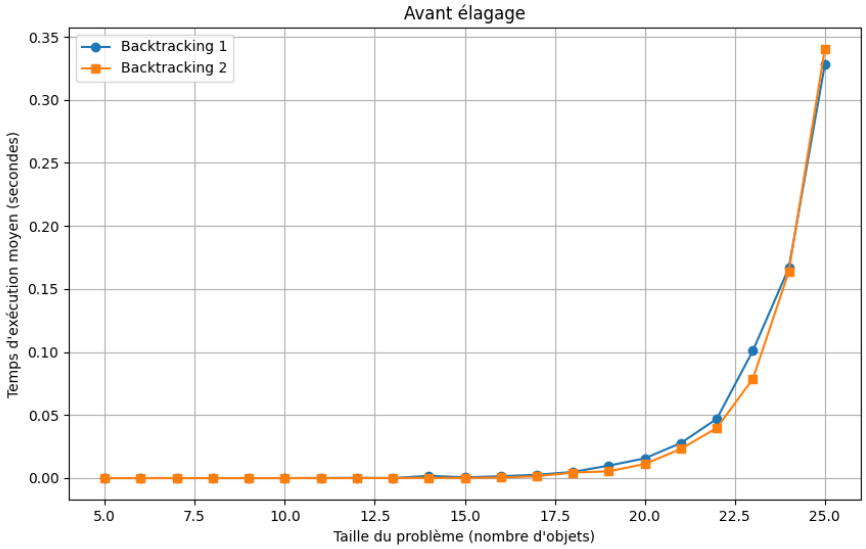
Compléxité temporelle : La complexité temporelle de cette version est exponentielle, car à chaque appel récursif, elle explore deux options : inclure ou ne pas inclure l'objet actuel, ce qui donne une complexité de $O(2^n)$, où n est le nombre d'objets.
Utilisation de la mémoire : les données sont partagées entre toutes les instances de l'algorithme,ce qui peut poser des problèmes de concurrence et de lisibilité du code. De plus, l'utilisation de variablesglobales peut rendre le code moins modulaire et plus difficile à maintenir.
Variables globales : L'utilisation de variables globales peut rendre le code plus difficile à comprendre et à maintenir, car elles peuvent être modifiées de manière imprévisible par d'autres parties du programme. Cela peutrendre le code plus sujet aux bugs et aux erreurs.

1.2.2 Backtracking 2 : sans variables globales

Compléxité temporelle : La complexité temporelle de cette version est également exponentielle, car elle explore également toutes les combinaisons possibles d'objets à inclure dans le sac à dos. Par conséquent, la complexité temporelle est également $O(2^n)$.
Utilisation de la mémoire : Cette version n'utilise pas de variables globales, mais passe plutôt les données nécessaires comme paramètres aux fonctions. Cela rend le code plus modulaire et plus facile à comprendre, car les données sont encapsulées dans les fonctions et ne sont pas partagées entre différentes parties du programme.
Variables locales : En utilisant des variables locales plutôt que des variables globales, cette version améliore la lisibilité et la maintenance du code en réduisant les effets de bord et en rendant les dépendances plus explicites.

1.3 Question 3 : Limite de performance de l'algorithme de backtracking

L'algorithme de *backtracking* devient trop lent à partir de 20 et c'est ce qu'on peut voir dans ce graphe :



2 Exercice 2 : Meilleur Backtracking

2.1 **Question 4 :** Amélioration du backtracking par élagage

2.1.1 Logique d'élagage :

Lors de l'exploration de chaque branche de l'arbre de recherche, si la borne supérieure calculée pour cette branche est inférieure à la meilleure solution actuellement trouvée, nous abandonnons l'exploration de cette branche. Cela nous permet d'éviter de manière proactive les branches qui ne peuvent pas conduire à une solution optimale, voir le code source des lignes 29 à 101 du fichier `knapstack.c`.

2.1.2 Efficacité de la stratégie d'élagage :

La stratégie d'élagage s'est révélée efficace pour réduire le temps de calcul de l'algorithme de backtracking. Les expérimentations ont montré une amélioration significative des performances, notamment en termes de temps d'exécution, lorsque la logique d'élagage est intégrée. Cela est corroboré par l'observation des données expérimentales, comme illustré dans le graphique associé : (voir unité de l'axe des abscisses)

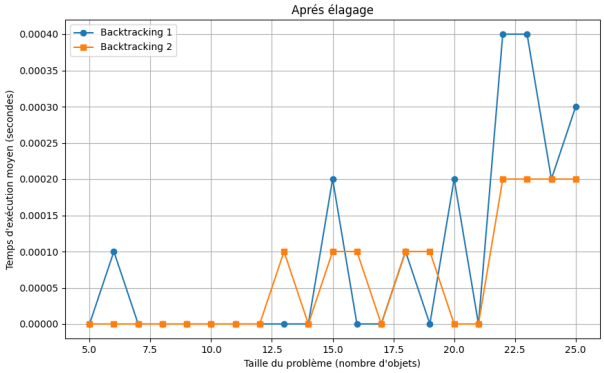


FIGURE 1 – Légende de l'image 1.

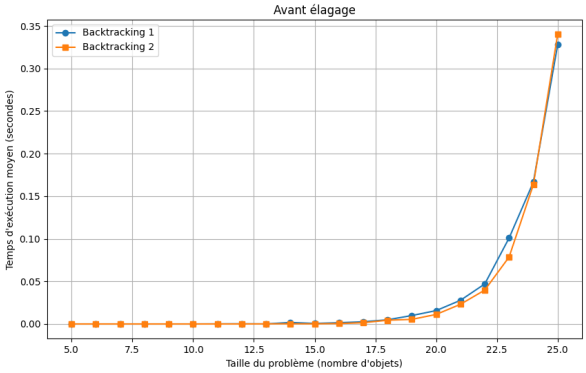


FIGURE 2 – Légende de l'image 2.

3 Exercice 3 : Programmation dynamique basée sur la valeur

3.1 Question 5 : Propriété de la sous-structure optimale

3.1.1 Propriété de la sous-structure optimale :

Considérons une instance du problème du sac à dos où W est la capacité maximale du sac et $O = \{o_1, o_2, ..., o_n\}$ notre ensemble d'objets. Chaque poids est accessible via w_i et chaque valeur avec v_i, où i est l'indice de l'objet dans l'ensemble O. Définissons $V = \sum_{i=1}^{n} v_i$ comme la somme des valeurs de tous les objets.
Supposons $S = \{o_{i_1}, o_{i_2}, ..., o_{i_k}\}$ est une solution optimale pour le problème $P(V)$, où o_{i_j} représente les objets choisis dans O pour former la solution S.
Alors, si nous retirons un objet o_i de S, noté $S' = S\backslash\{o_i\}$, cela signifie que S' est une solution optimale pour le sous-problème $P(V - v_i)$, où v_i est la valeur de l'objet o_i retiré de S.

Détaillons cette affirmation par l'absurde :
Supposons que $S = \{o_{i_1}, o_{i_2}, ..., o_{i_k}\}$ est une solution optimale pour $P(V)$, mais S' n'est pas une solution optimale pour $P(V - v_i)$. Cela implique qu'il existe une autre solution S'' pour $P(V - v_i)$ telle que la valeur totale de S'' est supérieure à celle de S'.
- Comme S est une solution optimale pour $P(V)$, la valeur totale de S est maximale parmi toutes les solutions pour $P(V)$. Par conséquent, en retirant l'objet o_i de S, nous obtenons S', qui est une solution pour $P(V - v_i)$.
Maintenant, si S' n'était pas une solution optimale pour $P(V - v_i)$, cela signifierait qu'il existe une solution S'' pour $P(V - v_i)$ avec une valeur totale encore plus grande. Cependant, cela contredit le fait que S était déjà une solution optimale pour $P(V)$, car cela impliquerait que S'' est également une solution pour $P(V)$, ce qui contredit la supposition initiale.
Par conséquent, la supposition initiale doit être vraie, et ainsi, si $S = \{o_{i_1}, o_{i_2}, ..., o_{i_k}\}$ est une solution optimale pour $P(V)$, alors $S' = S\backslash\{o_i\}$ est une solution optimale pour le sous-problème $P(V - v_i)$.

3.1.2 Definition récursive pour dp[v]

Pour chaque valeur cumulative v allant de 0 à V, nous définissons dp[v] comme le poids minimal nécessaire pour atteindre la valeur v.

La relation de récurrence pour remplir la table dp est donnée par :

$$dp[v] = \begin{cases} 0 & \text{si } v = 0 \\ \min(dp[v], dp[v - v(i)] + w(i)) & \text{si } v \geq v(i) \\ dp[v] & \text{sinon} \end{cases}$$

En résumé, pour chaque objet, nous avons deux choix :
— Ne pas inclure l'objet : la valeur reste celle de la solution sans cet objet, c'est-à-dire dp[v].
— Inclure l'objet : nous ajoutons le poids de cet objet à la solution optimale pour la valeur restante $v - v(i)$, c'est-à-dire dp[v - v(i)] + w(i), à condition que la valeur de l'objet ne dépasse pas v.

3.2 Question 6 : ajout de knapsackDP_Value

3.2.1 knapsackDP_Value :

```
1 int knapsackDP_Value(int W, Item items[], int n){
2     int V = 0, i,v;
3     int *dp;
4     for(i = 0; i < n; i++) {V += items[i].value;}
```

```
    dp = malloc((V+1) * sizeof(int));
    if(dp == NULL){ perror("erreur d'allocation pour dp"); exit(2);}
    dp[0] = 0;
    for(i = 1; i <= V; i++){ dp[i] = INT_MAX;}

    for(i = 0; i < n; i++)
        for(v = V; v > items[i].value; v--)
            dp[v] = min(dp[v], dp[v-items[i].value] + items[i].weight);
    int max_value = 0;
    for(i = V; i > 0; i--){
        if(dp[i] <= W ){
            max_value = i;
            break;
        }
    }
    free(dp);
    return max_value;
}
```

3.2.2 Compléxité temporelle de cette fonction :

La première boucle parcourt tous les objets une fois, donc elle a une complexité de $O(n)$. La deuxième boucle parcourt une plage de valeurs de V à 0 pour chaque objet, où V est la somme des valeurs de tous les objets. Ainsi, la complexité de cette boucle dépend de V. Dans le pire cas, V peut être aussi grand que la somme de toutes les valeurs des objets. Par conséquent, la complexité totale de l'algorithme est de $O(nV)$.

3.2.3 Graphes :

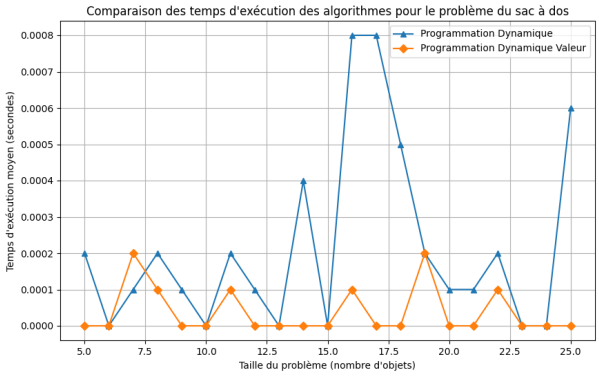


FIGURE 3 – Légende de l'image 1.

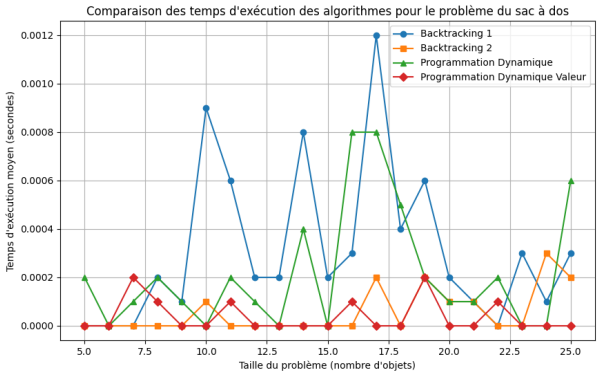


FIGURE 4 – Légende de l'image 2.

4 Exercice 4 : Sac à Dos 0/1 avec Poids Minimum

4.1 Question 7 : Conception d'un Algorithme de Programmation Dynamique

4.1.1 Definition récursive de la table dp :

La valeur stockée dans $dp[i][w]$ représente la valeur maximale obtenue en considérant les i premiers objets et en ayant un poids total de w. Cela signifie que $dp[i][w]$ contient la meilleure valeur que nous pouvons obtenir en choisissant parmi les premiers i objets sans dépasser le poids w.
on peut définir la table dp récursivement comme suit :

$$dp[i][w] = \begin{cases} \max(V(i-1) + dp[i-1][w - W(i-1)], dp[i-1][w]) & \text{si } W(i-1) \leq w \\ dp[i-1][w] & \text{sinon} \end{cases}$$

où $V(i)$ représente la valeur de l'objet i et $W(i)$ représente le poids de l'objet i.
En résumé, pour chaque objet, nous avons deux choix :

— Ne pas inclure l'objet : la valeur est celle de la solution sans cet objet.
— Inclure l'objet : nous ajoutons la valeur de cet objet à la meilleure solution possible pour le poids restant (c'est-à-dire, le poids total w moins le poids de l'objet inclus).

4.1.2 knapsackDP_min

Pour résoudre le problème du sac à dos 0/1 avec un poids minimum requis, nous suivons une approche similaire à celle du problème classique du sac à dos, avec quelques ajustements pour gérer la contrainte de poids minimum. Voici comment nous procédons :

1. **Initialisation de la table de programmation dynamique**
   Nous allouons une table dp de dimensions $(n+1) \times (W+1)$, où n est le nombre d'objets et W est la capacité maximale du sac à dos. Chaque entrée $dp[i][w]$ représente la valeur maximale obtenue en considérant les i premiers objets avec un poids total ne dépassant pas w.
2. **Remplissage de la table dp**
   Nous remplissons la table dp de la même manière que dans la fonction classique `knapsackDP`. Pour chaque objet, nous avons deux choix :
   — Ne pas inclure l'objet : la valeur reste celle de la solution sans cet objet, c'est-à-dire $dp[i-1][w]$.
   — Inclure l'objet : nous ajoutons la valeur de cet objet à la meilleure solution possible pour le poids restant, c'est-à-dire $dp[i-1][w-w(i)] + v(i)$, à condition que le poids de l'objet ne dépasse pas w.
3. **Recherche de la valeur maximale respectant la contrainte de poids minimum**
   Une fois la table dp remplie, nous devons trouver la valeur maximale qui satisfait la contrainte de poids minimum M. Nous parcourons la dernière ligne de la table $dp[n][w]$ pour w allant de M à W et sélectionnons la valeur maximale.
4. **Libération de la mémoire allouée**
   Enfin, nous libérons la mémoire allouée pour la table dp pour éviter les fuites de mémoire.

La fonction complète est donnée ci-dessous :

```
1  int knapsackDP_min(int W, int M, Item items[], int n) {
2      int i, w;
3      int **dp = (int**)malloc((n + 1) * sizeof(int*));
4      for(i = 0; i <= n; i++)
5          dp[i] = (int*)malloc((W + 1) * sizeof(int));
6
7      for(i = 0; i <= n; i++) {
8          for(w = 0; w <= W; w++) {
9              if (i == 0 || w == 0)
10                 dp[i][w] = 0;
11             else if (items[i-1].weight <= w)
12                 dp[i][w] = max(items[i-1].value + dp[i-1][w - items[i-1].
                       weight], dp[i-1][w]);
13             else
14                 dp[i][w] = dp[i-1][w];
15         }
16     }
17     int result = 0;
18     for (w = M; w <= W; w++) {
```

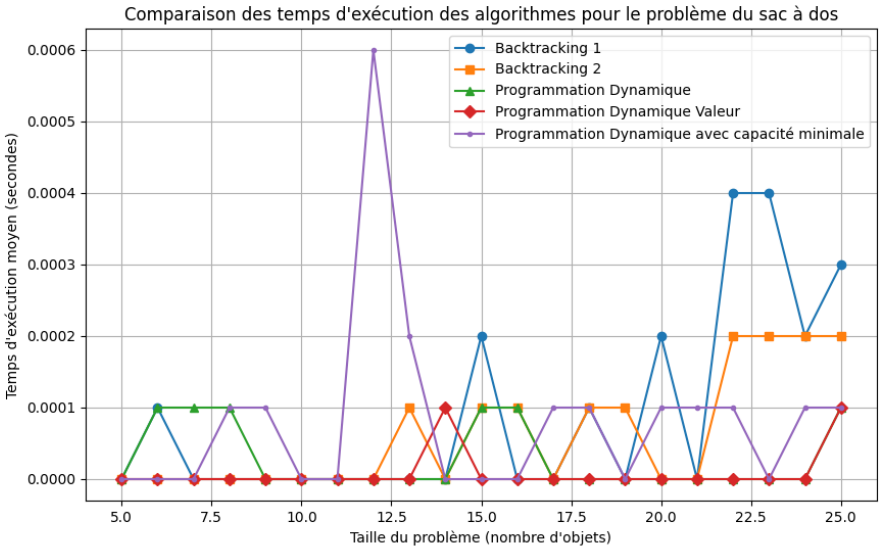
```
        if (dp[n][w] > result) {
            result = dp[n][w];
        }
    }
    for(i = 0; i <= n; i++)
        free(dp[i]);
    free(dp);

    return result;
}
```

4.2 Complexité spatiale et temporelle

La complexité spatiale de cette solution est de $O(n \times W)$, où n est le nombre d'éléments dans le tableau et W est la capacité maximale du sac à dos. Cela est dû à la mémoire allouée pour la matrice de programmation dynamique, qui a n lignes et W colonnes.
La complexité temporelle de cette solution est de $O(n \times W)$, où n est le nombre d'éléments dans le tableau et W est la capacité maximale du sac à dos. Cela est dû au calcul des valeurs dans la matrice de programmation dynamique, qui nécessite deux boucles imbriquées parcourant la matrice.

4.3 Graphes :

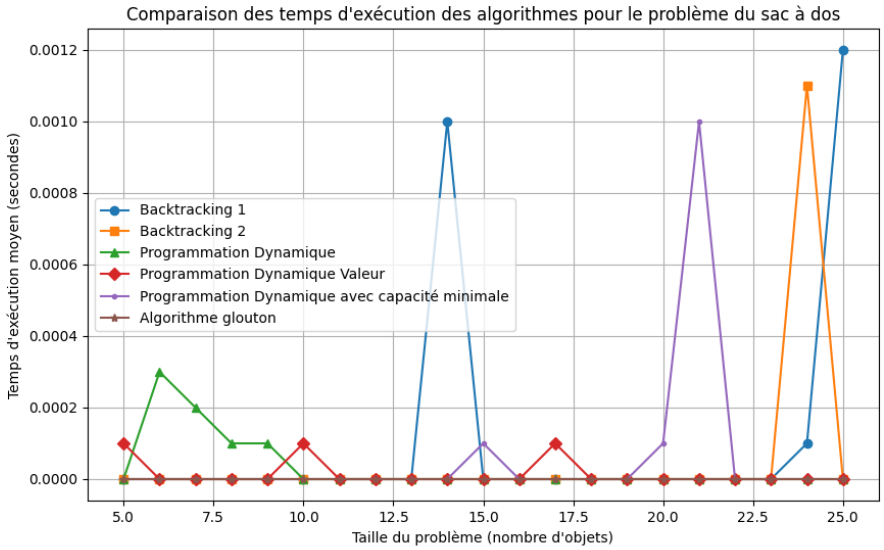


5 Exercice 5 : Approximation par un algorithme glouton

5.1 knapsack_greedy

```
int knapsack_greedy(int W, Item items[], int n){
    //on trie le tableau par ordre decroissant
    tri_fusion(items, 0, n-1);
    //on ajoute les elements
    int i=0, total_weight = 0, best_value = 0;
    for(i = 0; i < n; i++){
        if((total_weight + items[i].weight) <= W)
            total_weight += items[i].weight;
            best_value += items[i].value;
    }
    return best_value;
}
```

5.2 Graphes



5.3 Exemple où l'algorithme glouton n'est pas optimal

Considérons l'exemple suivant :

— Objet 1 : valeur = 40, poids = 10
— Objet 2 : valeur = 100, poids = 50
— Capacité du sac à dos = 50

L'algorithme glouton choisira l'objet 1 (rapport = 4), puis il ne pourra plus ajouter l'objet 2, donc la valeur totale sera 40. Cependant, l'optimal est de choisir l'objet 2 seul avec une valeur totale de 100.

5.4 Complexité temporelle et spatiale

Complexité temporelle :

— Tri des objets par rapport décroissant : $O(n \log n)$
— Sélection des objets : $O(n)$

Donc, la complexité temporelle totale est $O(n \log n)$.

Complexité spatiale : L'espace utilisé par l'algorithme est principalement pour stocker les objets et leurs rapports, ce qui est $O(n)$.

En résumé, l'algorithme glouton est simple et rapide à implémenter avec une complexité temporelle de $O(n \log n)$, mais il peut ne pas toujours donner la solution optimale pour certaines instances spécifiques.

6 Conclusion

Nous avons exploré diverses méthodes pour résoudre le problème du sac à dos 0/1. Nous avons d'abord analysé des algorithmes de backtracking et amélioré leurs performances en introduisant une logique d'élagage. Ensuite, nous avons étudié la programmation dynamique, notamment une version basée sur les valeurs et une autre assurant un poids minimum requis. Enfin, nous avons implémenté une solution d'approximation par un algorithme glouton, tout en reconnaissant ses limitations. Chaque approche a ses avantages et inconvénients, mais ensemble, elles offrent une compréhension complète des stratégies disponibles pour ce problème classique.