République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université des Sciences et de la Technologie Houari Boumediene

Faculté d'Informatique

Département d'Intelligence Artificielle et Sciences des Données

Partie 1

Recherche basée espace des états

MODULE : Métaheuristique

Réalisé par :

TAOURIRT Hamza

SAADA Samir

Année Universitaire : 2023/2024

TABLE DES MATIERES

1. Introduction

Le Problème des Sacs à Dos Multiples (Multiple Knapsack Problem - MKP) représente un défi fondamental en optimisation combinatoire. Il trouve des applications dans de nombreux domaines, notamment la logistique, la gestion des ressources, et la planification de projets. L'essence du problème réside dans la maximisation de la valeur totale des objets sélectionnés, tout en respectant les contraintes de capacité de plusieurs sacs à dos de capacités différentes.

Dans le cadre de ce projet, nous nous penchons sur la résolution du MKP à travers diverses méthodes, de la recherche exhaustive aux techniques heuristiques avancées. Notre objectif est de modéliser le problème, de développer des approches de résolution efficaces, d'expérimenter avec différentes tailles de problèmes, et enfin de comparer les performances de ces approches.

Ce rapport propose une exploration approfondie du MKP. Dans la première partie, nous introduisons formellement le problème, détaillons ses composantes et ses contraintes. Ensuite, nous présentons les méthodes exactes telles que la recherche en largeur d'abord (BFS) et la recherche en profondeur d'abord (DFS) pour résoudre le problème. Nous examinons également les aspects théoriques et algorithmiques de ces approches, illustrés par des exemples.

Dans la deuxième partie, nous explorons les méthodes heuristiques, en mettant particulièrement l'accent sur l'algorithme A* pour le MKP. Nous définissons une fonction heuristique et une fonction de coût appropriées.

La troisième partie de ce rapport présente nos expérimentations. Nous avons réalisé des tests avec différentes tailles de problèmes pour chaque approche, collecté des données sur le nombre de nœuds développés, les temps d'exécution et les performances obtenues. Ces expérimentations fournissent un aperçu des performances relatives de chaque méthode.

Enfin, dans la dernière partie, nous comparons les résultats obtenus et discutons des avantages et des limitations de chaque approche. Nous mettons en évidence les tendances observées et offrons des perspectives sur les pistes d'amélioration et les domaines d'application potentiels.

1.1 Définition (Multiple Knapsack Problem - MKP)

Le Problème des Sacs à Dos Multiples (Multiple Knapsack Problem - MKP) est une généralisation du Problème du Sac à Dos (Knapsack Problem - KP), un problème d'optimisation combinatoire. Dans le KP, l'objectif est de maximiser la valeur totale des objets placés dans un unique sac à dos, tout en respectant sa capacité maximale. En revanche, dans le MKP, le défi est étendu à plusieurs sacs à dos, chacun ayant sa propre capacité. Ainsi, dans le MKP, nous sommes chargés de répartir un ensemble prédéfini d'objets entre ces sacs de façon à maximiser la somme des valeurs des objets sélectionnés, tout en observant les contraintes de capacité spécifiques à chaque sac.

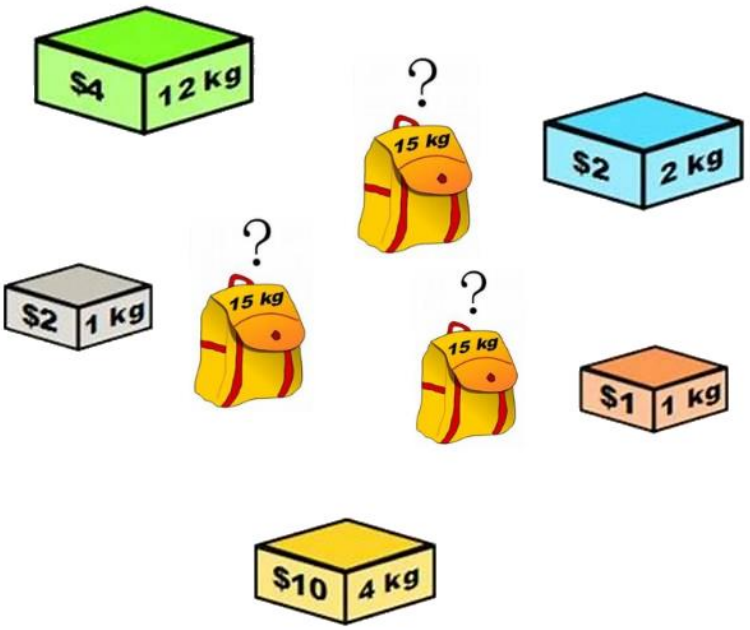

Figure 1 : Le Problème des Sacs à Dos Multiples.

Ci-dessous est une représentation mathématique du MKP :

Maximiser la somme des bénéfices des objets alloués aux sacs à dos :

$$\max \sum_{i=1}^{m} \sum_{j=1}^{n} p_j x_{ij}$$

Sous les contraintes suivantes :

Chaque objet (j) peut être attribué à au plus un sac à dos (i) :

$$\sum_{i=1}^{m} x_{ij} \leq 1, \quad \forall j = 1, \ldots, n$$

La somme des poids des objets dans chaque sac à dos (i) ne doit pas dépasser sa capacité (c_i) :

$$\sum_{j=1}^{n} w_j x_{ij} \leq c_i, \quad \forall i = 1, \ldots, m$$

Les variables (x_{ij}) sont binaires (1) ou non (0), indiquant si l'objet (j) est attribué au sac à dos (i) :

$$x_{ij} \in \{0, 1\}, \quad \forall i = 1, \ldots, m, \quad \forall j = 1, \ldots, n$$

1.2. Exemple Sacs à Dos Multiples

Considérons un exemple illustratif du MKP :

Dans cet exemple du Problème des Sacs à Dos Multiples (MKP), nous avons trois sacs à dos avec les capacités suivantes : $C_1 = 10$, $C_2 = 15$, $C_3 = 20$. Nous disposons également de cinq objets, chacun ayant une valeur et un poids spécifiques :

Objet	Valeur	Poids
1	8	4
2	10	6
3	6	3
4	3	4
5	7	13

Tableau 1 : Tableau d'exemple du Problème des Sacs à Dos Multiples (MKP).

L'objectif est de répartir ces cinq objets entre les trois sacs à dos de manière à maximiser la valeur totale des objets tout en respectant les capacités de chaque sac.

Une solution possible consiste à placer l'objet 1 dans le sac à dos 1, les objets 2 et 3 dans le sac à dos 2, et les objets 4 et 5 dans le sac à dos 3. Cette allocation donne une valeur totale de 8 + 10 + 6 + 3 + 7 = 34, respectant ainsi les contraintes de capacité de chaque sac.

2. Résolution à l'aide des méthodes exactes

2.1. Fonctionnement de la méthode exacte Breadth-First Search

Breadth-First Search (BFS) est un algorithme de parcours de graphe utilisé pour explorer de manière systématique les nœuds d'un graphe. Il commence par explorer tous les voisins du nœud actuel avant de passer aux voisins des voisins. L'algorithme BFS est généralement utilisé pour trouver le plus court chemin entre deux nœuds dans un graphe non pondéré. Cependant, il peut également être appliqué à d'autres types de problèmes, y compris la résolution du Problème des Sacs à Dos Multiples (MKP), lorsque la recherche d'une solution exacte est nécessaire.

Dans le cadre du MKP, BFS est utilisé pour explorer de manière exhaustive toutes les combinaisons possibles d'objets placés dans les sacs à dos. Voici comment fonctionne BFS pour résoudre le MKP :

- Initialisation :
  - Commencez par un état initial où aucun objet n'est placé dans aucun sac à dos.
  - Créez une file d'attente pour stocker les états à explorer.
- Boucle principale :

  Tant que la file d'attente n'est pas vide :
  - Retirez un état de la file d'attente.
  - Pour cet état, générez tous les états voisins en ajoutant un objet non placé dans chaque sac à dos possible.
  - Vérifiez chaque nouvel état généré :

- Si l'état est valide (c'est-à-dire qu'il respecte les contraintes de capacité de chaque sac à dos), ajoutez-le à la file d'attente pour une exploration ultérieure.
- Si l'état est une solution optimale (tous les objets sont placés et la valeur totale est maximisée), mettez à jour la meilleure solution trouvée jusqu'à présent.

➢ Arrêt :

- L'algorithme s'arrête lorsque la file d'attente est vide, ce qui signifie que tous les états possibles ont été explorés.

➢ Retour de la solution :

- Retournez la meilleure solution trouvée, c'est-à-dire la configuration d'objets qui maximise la valeur totale tout en respectant les contraintes de capacité des sacs à dos.

2.2. Algorithme de la méthode (BFS)

```
Procédure BFS_MKP ()
Entrée : Capacités des sacs, Liste d'objets, Zone de résultats, Zone
         de métriques
Variables : Sacks : Liste de listes d'objets, Capacités : Liste de
            capacités des sacs, Liste d'objets à placer dans les
            sacs, État : État courant dans l'arbre de recherche,
            ProfondeurMax : Profondeur maximale de l'arbre de
            recherche, File : File pour la stratégie BFS, Graph :
            Graphe pour la visualisation
DEBUT
      Initialiser Sacks avec des sacs vides
      Initialiser l'état initial avec des sacs vides, une capacité
      totale de 0, et un index d'objet de départ
      Enfiler l'état initial dans une file
      Initialiser le nombre de nœuds explorés à 0
```

```
    Tant que (la file n'est pas vide et la profondeur maximale
      n'est pas atteinte) Faire
       État = Premier élément de la file
       Défiler la file
       Incrémenter le nombre de nœuds explorés
       Si (État.Profondeur < ProfondeurMax) Alors
          Pour chaque objet dans la liste d'objets Faire
             Si l'objet peut être placé dans un des sacs en
               respectant les capacités Alors
                Créer un nouvel état en plaçant l'objet dans
                un des sacs
                Mettre à jour la file en ajoutant le nouvel
                état
             Fin Si
          Fin Pour
       Fin Si
    Fin Tant que
    Générer le fichier DOT pour visualiser l'arbre de recherche
    Afficher les métriques
Fin
```

2.3. Exemple d'exécution de l'Algo

Prenons un exemple concret avec des objets ayant des poids et des valeurs spécifiques, ainsi que deux sacs à dos avec une capacité de 40 unités chacun.

Supposons que nous ayons trois objets avec les poids suivants : 10, 10 et 30, et les valeurs correspondantes : 60, 100 et 120. Nous souhaitons déterminer la meilleure combinaison d'objets à placer dans les deux sacs à dos pour maximiser la valeur totale. Maintenant, voyons comment cela peut être démontré graphiquement en représentant

les différents états explorés par l'algorithme et en mettant en évidence la progression de la recherche vers la solution optimale :

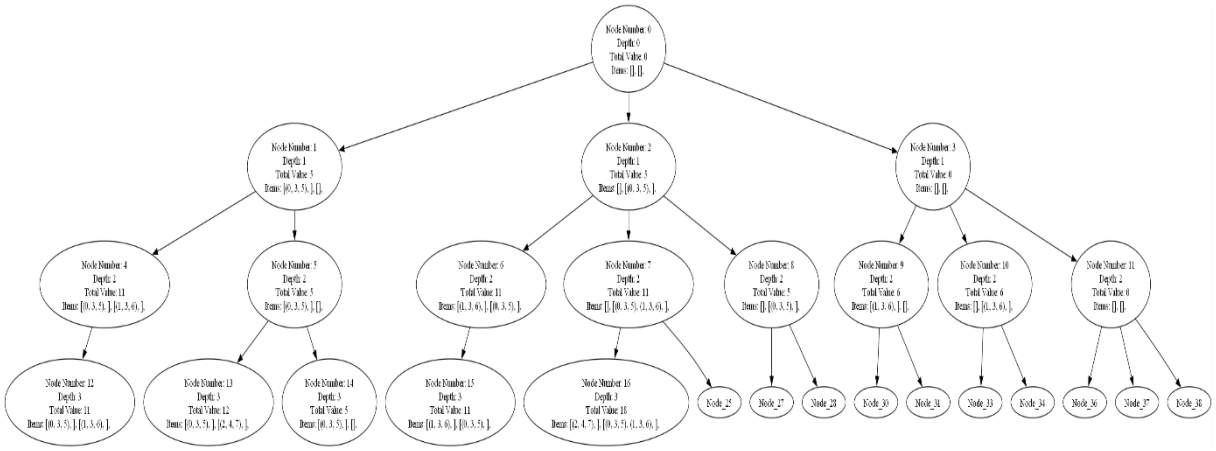

Figure 2 : Exploration BFS dans le Multi-Knapsack Problem « Généré par notre Application ».

L'algorithme BFS commence par un état initial où aucun objet n'est placé dans aucun sac à dos. Ensuite, il génère tous les états possibles en ajoutant un objet non placé dans chaque sac à dos. Par exemple, dans notre cas, il pourrait générer des états où le premier objet est placé dans le premier sac à dos, ou dans le deuxième sac à dos, ou n'est pas placé du tout. Il continue à explorer tous les états possibles de cette manière.

À chaque itération, l'algorithme BFS évalue les états générés pour vérifier s'ils respectent les contraintes de capacité des sacs à dos et s'ils constituent une solution optimale. Il met à jour la meilleure solution trouvée jusqu'à présent en fonction de la valeur totale des objets placés dans les sacs à dos.

En fin de compte, une fois que tous les états possibles ont été explorés et que la file d'attente est vide, l'algorithme retourne la meilleure solution trouvée, c'est-à-dire la combinaison d'objets qui maximise la valeur totale tout en respectant les contraintes de capacité des sacs à dos.

2.4 Fonctionnement de la méthode exacte Depth-First Search (DFS)

La méthode Depth-First Search (DFS), ou recherche en profondeur, est une méthode de recherche qui explore autant que possible le graphe en allant aussi loin que possible le long d'une branche avant de revenir en arrière. Dans le contexte du Problème des Sacs à Dos Multiples (MKP), la DFS est utilisée pour explorer toutes les combinaisons possibles d'objets placés dans les sacs à dos.

- Initialisation : La recherche commence par initialiser une pile de nœuds à explorer. Le premier nœud ajouté à la pile est l'état initial du problème.
- Exploration : À chaque itération, le nœud le plus récemment ajouté à la pile est retiré et ses enfants potentiels sont générés. Pour chaque enfant potentiel, la DFS vérifie s'il est réalisable (c'est-à-dire si la capacité des sacs n'est pas dépassée) et s'il conduit à une meilleure solution. Si c'est le cas, l'enfant est ajouté à la pile.
- Backtracking : Lorsque la DFS atteint un nœud sans enfants réalisables ou que tous les nœuds ont été explorés, elle revient en arrière (backtrack) au nœud précédent pour explorer d'autres options.
- Arrêt : Le processus se poursuit jusqu'à ce que tous les nœuds aient été explorés ou qu'une solution satisfaisante soit trouvée.
- Retour de la meilleure solution : Une fois que la recherche est terminée, la meilleure solution trouvée est retournée.

La DFS peut être récursive ou utilisée de manière itérative avec une pile pour garder trace des nœuds à explorer. Cette méthode est souvent utilisée pour explorer de grands espaces d'états et est particulièrement efficace lorsque la profondeur de recherche est limitée et qu'il existe une forte corrélation entre la profondeur et la qualité de la solution.

2.5 Algorithme de la méthode (DFS)

```
Procédure DFS_MKP ()
Entrée : Capacités des sacs, Liste d'objets, Zone de résultats, Zone de
         métriques
Variables : Sacks : Liste de listes d'objets, Capacités : Liste de
            capacités des sacs, Objets : Liste d'objets à placer dans les
            sacs, État : État courant dans l'arbre de recherche,
            ProfondeurMax : Profondeur maximale de l'arbre de recherche ,
            Graph : Graphe pour la visualisation
DEBUT
    Initialiser Sacks avec des sacs vides
    Initialiser l'état initial avec des sacs vides, une capacité totale
    de 0, et un index d'objet de départ
```

```
    Initialiser le nombre de nœuds explorés à 0
    Tant que (la pile n'est pas vide et la profondeur maximale n'est
        pas atteinte) faire

      État = Sommet de la pile
      Dépiler la pile
      Incrémenter le nombre de nœuds explorés
      Si (État.Profondeur < ProfondeurMax) alors
        Pour chaque objet dans la liste d'objets Faire
          Si l'objet peut être placé dans un des sacs en
            respectant les capacités alors
            Créer un nouvel état en plaçant l'objet dans un des
            sacs
            Mettre à jour la pile en ajoutant le nouvel état
          Fin Si
        Fin Pour
      Fin Si
    Fin Tant que
    Générer le fichier DOT pour visualiser l'arbre de recherche
    Afficher les métriques
Fin
```

2.6 Exemple d'exécution de l'Algo DFS

Prenons un exemple illustratif en utilisant le graphe suivant pour représenter l'exécution de l'algorithme DFS sur l'instance donnée du problème du Multi-Knapsack (*figure 3*). Supposons que nous ayons trois objets avec des poids de 10, 10 et 30, et des valeurs de 60, 100 et 120 respectivement. Nous disposons également de deux sacs à dos ayant une capacité de 40 unités chacun.

En appliquant DFS, nous débutons par placer un objet dans le premier sac à dos, puis continuons à explorer toutes les combinaisons possibles de placement jusqu'à ce que le premier sac à dos atteigne sa capacité maximale ou que tous les objets soient placés. Ensuite, DFS poursuit en plaçant des objets dans le deuxième sac à dos et en explorant à nouveau toutes les combinaisons possibles. Cette approche permet

d'examiner de manière exhaustive toutes les configurations d'objets dans les sacs à dos jusqu'à ce qu'une solution optimale soit trouvée ou que toutes les possibilités aient été explorées.

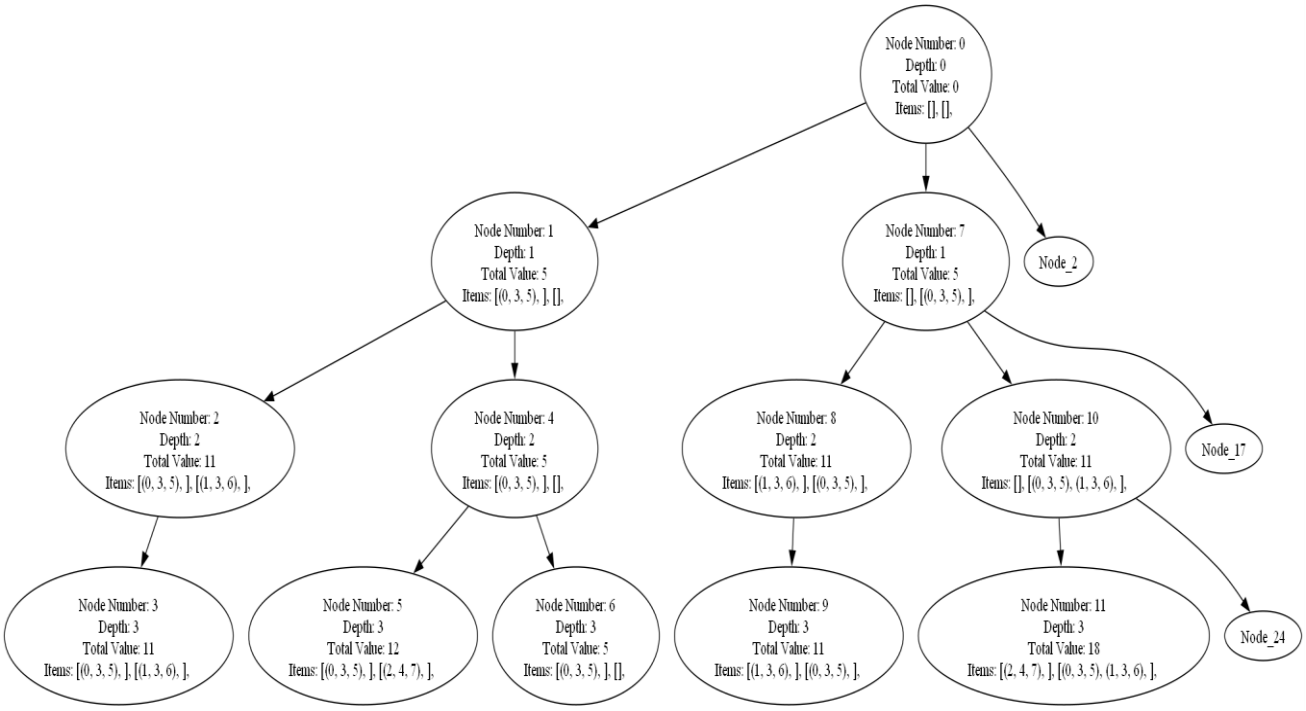

Figure 3 : Exploration DFS dans le Multi-Knapsack Problem « Généré par notre Application ».

2.7 Fonctionnement de la méthode A*

Le problème du sac à dos est un problème d'optimisation classique dans lequel des objets de différents poids et valeurs doivent être placés dans des sacs à dos de capacité limitée, de manière à maximiser la valeur totale des objets tout en respectant les contraintes de capacité. L'heuristique A* est une méthode de recherche qui combine un coût réel g(n) et une estimation heuristique h(n) pour guider la recherche vers la solution optimale de manière efficace. Dans ce rapport, nous examinerons en détail l'heuristique utilisée dans l'algorithme A* pour résoudre le problème du sac à dos.

Définition Mathématique de l'Heuristique

Définissons les variables suivantes :

n : le nombre d'articles m : le nombre de sacs à dos c_j : la capacité du sac à dos j v_i : la valeur de l'article i w_i : le poids de l'article i x_ij : une variable binaire indiquant si l'article i est attribué au sac à dos j (1) ou non (0) S : l'ensemble des articles non encore attribués à aucun sac à dos

La fonction heuristique h peut être formulée comme suit :

$$\text{Maximize:} \quad h = \max\left\{\sum_{i \in S} v_i \cdot x_{ij}\right\}$$

$$\text{Subject to:} \quad \sum_{i=1}^{n} w_i \cdot x_{ij} \leq c_j, \quad \text{for all } j = 1, 2, \ldots, m$$

$$\sum_{j=1}^{m} x_{ij} \leq 1, \quad \text{for all } i = 1, 2, \ldots, n$$

$$x_{ij} \in \{0, 1\}, \quad \text{for all } i = 1, 2, \ldots, n \text{ and } j = 1, 2, \ldots, m$$

$$i \in S$$

Dans cette formulation, la fonction heuristique h vise à maximiser la valeur totale des articles pouvant être attribués aux capacités restantes des sacs à dos, en respectant les contraintes suivantes :

- La somme des poids des articles attribués à chaque sac à dos j ne doit pas dépasser sa capacité c_j.
- Chaque article i peut être attribué à au plus un sac à dos.

- Les variables de décision x_ij sont binaires, indiquant si un article i est attribué à un sac à dos j ou non.
- La maximisation est effectuée uniquement sur l'ensemble S des articles non encore attribués à aucun sac à dos.

La fonction heuristique h fournit une borne supérieure sur la valeur maximale supplémentaire pouvant être obtenue en attribuant les articles restants aux sacs à dos, en tenant compte de leurs capacités et des ratios de profit/poids des articles.

Cette formulation est elle-même un problème de sac à dos, mais elle est résolue de manière gloutonne en triant les articles par leurs ratios de profit/poids et en les attribuant aux sacs à dos dans cet ordre, tant que les contraintes de capacité sont satisfaites.

Notez que cette fonction heuristique est admissible (c'est-à-dire qu'elle ne surestime jamais la valeur réelle restante) car elle prend en compte les capacités restantes des sacs à dos et les ratios de profit/poids des articles, ce qui garantit que la valeur estimée est au plus égale à la valeur réelle pouvant être obtenue.

2.8 Algorithme de la méthode A*

```
Procédure AStar_MKP ()
Entrée : Capacités des sacs, Liste d'objets, Zone de résultats, Zone de
         métriques, Profondeur maximale
Variables : Sacks : Liste de listes d'objets, Capacités : Liste de
            capacités des sacs, Objets : Liste d'objets à placer dans les
```

```
        sacs, État : État courant dans l'arbre de recherche,
        FilePrioritaire : File de priorité pour l'algorithme A*, Graph
        : Graphe pour la visualisation, ProfondeurMax : Profondeur
        maximale de l'arbre de recherche
DEBUT
    Initialiser Sacks avec des sacs vides
    Initialiser l'état initial avec des sacs vides, une capacité totale
    de 0, et un index d'objet de départ
    Créer une file de priorité avec une fonction d'évaluation f = g + h
    Ajouter l'état initial à la file de priorité avec une évaluation
    initiale f = g + h
    Initialiser le nombre de nœuds explorés à 0
    Tant que la file de priorité n'est pas vide Faire
        État ← Retirer et sauvegarder le premier élément de la file
        de priorité
        Ajouter l'état à la liste de tous les états
        Sacs ← Sacs de l'état
        PoidsTotal ← Poids total de l'état
        IndexObjet ← Index de l'objet de l'état
        ObjectifAtteint ← objectifVal ≤ calculerValeurActuelle(Sacs)
        Mettre à jour la profondeur maximale
        Si la profondeur maximale est atteinte Alors
            Passer à l'itération suivant
        Fin si
        Calculer la valeur totale
        Si l'index de l'objet est dans les limites des objets Alors
            Afficher les sacs
            Calculer la valeur totale
        Fin Si
        Marquer l'état comme visité et lui attribuer un numéro de
        nœud Créer un nouvel état avec les mêmes sacs, poids total et
        index d'objet Ajouter le nouvel état à la liste de tous les
        sacs
        Si l'objectif est atteint alors
            arrêter la boucle
        Fin SI
```

```
            Si l'index de l'objet est dans les limites des objets Alors
                Pour chaque sac et objet Faire
                    Créer un nouvel état si l'objet peut être placé dans le
                    sac Ajouter le nouvel état à la file de priorité
                Fin Pour
            Fin Si
        Fin Tant que
Fin
```

2.9 Explication de l'Heuristique

Dans cette approche:

1. **Ordonnancement des Objets et des Sacs**: Les objets sont classés en fonction de leur densité de valeur, calculée comme le rapport entre la valeur de l'objet et son poids. Les sacs sont également ordonnés par capacité, du plus petit au plus grand.
2. **Évaluation de la Solution Actuelle (g(s))**: La fonction d'évaluation $g(s)$ mesure le coût de la solution actuelle. Elle représente la valeur totale des objets déjà placés dans les sacs.
3. **Estimation de la Solution Future (h(s))**: L'estimation de la solution future)$h(s)$ est basée sur une approche gloutonne. On itère sur les sacs en sélectionnant les objets les plus rentables (selon la densité de valeur) pour les placer dans les sacs jusqu'à ce que leur capacité soit épuisée. Cette estimation repose sur l'hypothèse que le remplissage optimal des sacs se fait en priorisant les objets les plus rentables.
4. **Fonction d'Évaluation Globale (f(s))**: La fonction d'évaluation globale $f(s)$ combine $g(s)$ et $h(s)$, représentant le coût actuel et l'estimation du coût futur. Mathématiquement, $f(s)=g(s)+h(s)$.

En priorisant les sacs avec une capacité restante suffisante pour accueillir les objets les plus rentables, l'heuristique A* vise à explorer efficacement l'espace de recherche pour atteindre une solution optimale ou proche de l'optimalité. La valeur de la solution obtenue représente le résultat de $f(s)$, fournissant ainsi une indication de la qualité de la solution trouvée par l'algorithme.

3. Analyse et Complexité des Algorithmes DFS, BFS et A*

L'efficacité des algorithmes de recherche, tels que la recherche en profondeur (DFS), la recherche en largeur (BFS) et l'algorithme A*, dans la résolution du problème du sac à dos multiple repose sur leur complexité. Dans cette section, nous analyserons la complexité en temps de ces algorithmes dans le pire des cas, en prenant en compte le nombre d'objets et de sacs.

La complexité en temps des algorithmes DFS, BFS et A* pour le problème du sac à dos multiple est exponentielle dans le pire des cas. Plus précisément, elle est de l'ordre b^n, où n est le nombre d'objets et b est le facteur de branchement (nombre de nœuds enfants). À chaque niveau de profondeur de l'arbre de recherche, nous devons décider si nous plaçons l'objet n dans l'un des sacs ou non. Cette décision est répétée pour chaque objet, ce qui signifie que chaque niveau de profondeur représente le choix de placer ou non le nième objet dans l'un des sacs. La croissance exponentielle de l'arbre de recherche est due à la nature combinatoire du problème. Chaque nouvel objet introduit b possibilités de placement (dans l'un des b sacs), ce qui conduit à une expansion exponentielle du nombre total de configurations possibles. Le nombre total de nœuds possibles dans l'arbre de recherche est $b^{(n+1)}$ -1. Car dans le premier niveau on doit décider si on place le sac dans l'un des sacs ou pas au deuxième niveau c'est au tour de l'objet deux qu'on doit placer dans les m sacs m fois, ce qui veux dire qu'on a une suite :

$$b + b * b + b * b * b + \cdots b^n$$

Donc une somme de termes où chaque terme consiste en b multiplié par lui-même un certain nombre de fois, le nombre de fois augmentant de 1 à n. Cela peut être représenté comme une série géométrique.

Autres Considérations

Il est important de noter que la complexité en temps mentionnée ci-dessus ne tient pas compte des heuristiques ou des optimisations spécifiques qui pourraient être appliquées dans le cadre de l'algorithme A*. En pratique, l'utilisation d'heuristiques appropriées peut réduire considérablement le nombre de nœuds explorés dans l'arbre de recherche, améliorant ainsi les performances de l'algorithme.

De plus, la complexité spatiale des algorithmes peut également être un facteur à prendre en compte, surtout lorsque le nombre d'objets et de sacs est important. Des stratégies de gestion de la mémoire peuvent être nécessaires pour éviter les problèmes de consommation excessive de mémoire.

La complexité exponentielle des algorithmes DFS, BFS et A* pour le problème du sac à dos multiple souligne les défis inhérents à la résolution de ce problème. Bien que ces algorithmes offrent des approches différentes pour explorer l'espace de recherche, ils sont tous confrontés à une croissance exponentielle du nombre de configurations à évaluer. Il est essentiel de prendre en compte cette complexité lors de la conception et de l'implémentation d'algorithmes efficaces pour résoudre le problème du sac à dos multiple.

4. Section Expérimentale

Configuration Expérimentale Les expériences ont été menées sur un ordinateur équipé des spécifications suivantes :

- Processeur : Intel(R) Atom(TM) x5-Z8350 CPU @ 1.44GHz avec une fréquence de 1.44 GHz.
- Mémoire RAM : 4.00 Go (3.83 Go utilisables).
- Environnement de Développement Intégré (IDE) : Eclipse 2023.
- Langage de Programmation : Java 21.

Processus Expérimental a suivi les étapes suivantes :

1. Génération des Données : Nous avons utilisé notre interface pour générer dix fichiers CSV différents contenant des paires poids-valeur générées aléatoirement. Les bornes supérieure et inférieure ont été sélectionnées manuellement à l'aide de l'interface.
2. Configuration des Paramètres : À l'aide de l'interface, nous avons choisi l'algorithme BFS, DFS ou A* ainsi que le nombre de sacs avec leurs capacités respectives. Les poids des objets et la capacité des sacs ont été sélectionnés de manière à éviter des résultats triviaux, en veillant à ce que les sacs puissent contenir un certain nombre d'objets mais pas tous.
3. Contraintes d'Arrêt : En raison de la complexité temporelle élevée de l'ordre de $((m)^n)$, nous avons défini des contraintes pour arrêter les algorithmes s'ils prennent trop de temps, comme une profondeur maximale, une durée maximale ou un nombre maximum de nœuds dans la pile (ou la file d'attente).
4. Collecte des Données : Les données collectées comprenaient la durée, le nombre de nœuds, la profondeur atteinte, le ratio valeur maximale/valeur cible pour chaque instance sur dix pour chaque DFS, BFS et A*. Ensuite, la durée moyenne et l'écart-type pour chaque exécution d'algorithme ont été collectés.

5. Analyse des Résultats : Les résultats ont été résumés dans des tableaux et des graphiques.

4.1 Résultats de l'expérimentation sur les algorithmes BFS, DFS et A*

Notre test a été réalisé sur une interface dédiée à tester ces algorithmes. Les paramètres sélectionnés étaient les suivants : les poids étaient attribués de manière aléatoire aux objets avec un poids minimum de 10 et une borne maximale de 20, les valeurs étaient également aléatoires avec une limite supérieure de 20 ; trois sacs ont été utilisés avec des capacités fixes de 25, 18 et 35. 10 fichiers CSV de paires poids-valeur représentant les objets ont été générés en utilisant l'interface. Le nombre le plus bas d'objets est de cinq et il augmente d'un jusqu'à atteindre 14.

La durée pour chaque exécution de chaque algorithme a été enregistré puis les moyennes et déviation standard de chaque algorithme calculé, les résultats son résumé dans le tableau et dans le graphe ci-dessous

Nombre d'objets	BFS	DFS	A*
5	0.222	0.023	0.247
6	1.738	1.91	2.004
7	2.013	2.91	3.106
8	3.223	4.061	4.634
9	2.395	3.741	3.986
10	7.298	9.317	10.322
11	18.384	22.107	28.047
12	25.462	34.839	37.641
13	27.011	39.276	46.055
14	106.092	127.723	150.884
Moyenne	**19.3838**	**24.5907**	**28.6926**
Deviation standard	**29.06197**	**35.21445**	**41.57742**

Tableau 2 : *Évaluation des algorithmes de sac à dos sur des configurations variables*

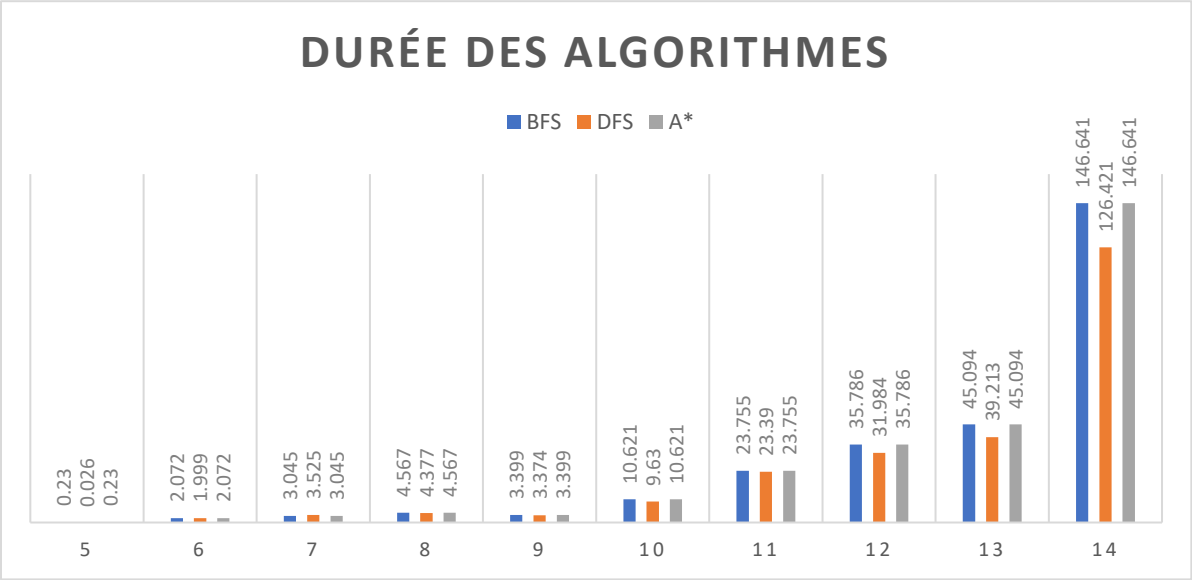

Figure 4 : Analyse comparative des performances des algorithmes BFS, DFS et A*

On peut analyser les performances des algorithmes de Recherche en Largeur (BFS), de Recherche en Profondeur (DFS) et A* dans la résolution du Problème de Sac à Dos Multiple (MKP) avec trois sacs.

Tendance générale : À mesure que le nombre d'objets augmente, la durée (en secondes) pour les trois algorithmes augmente également, ce qui est attendu car la complexité du problème croît avec plus d'objets. Comparaison des algorithmes : L'algorithme A* prend systématiquement plus de temps que BFS et DFS pour le nombre d'objets donné. DFS prend plus de temps que BFS dans la plupart des cas, sauf pour 9 objets. Les différences de performance entre les algorithmes deviennent plus prononcées à mesure que le nombre d'objets augmente. Durée moyenne : La durée moyenne (en secondes) pour chaque algorithme est : BFS : 19.3838 secondes DFS : 24.5907 secondes A* : 28.6926 secondes Cela montre qu'en moyenne, BFS est l'algorithme le plus rapide, suivi de DFS et ensuite A*.

Écart-type : L'écart-type (en secondes) pour chaque algorithme est : BFS : 29.06197 secondes DFS : 35.21445 secondes A* : 41.57742 secondes Les valeurs d'écart-type plus élevées indiquent une plus grande dispersion des durées sur différents nombres d'objets. A* a l'écart-type le plus élevé, suggérant que ses performances peuvent être plus sensibles au nombre d'objets par rapport à BFS et DFS.

Observations : Pour de plus petits nombres d'objets (jusqu'à 9), les différences de performance entre les algorithmes sont relativement faibles. À mesure que le nombre d'objets augmente (10 ou plus), les écarts de performance entre les algorithmes deviennent plus importants, A* prenant significativement plus de temps que BFS et DFS. L'augmentation significative des durées pour de plus grands nombres d'objets (par

exemple, 14 objets) suggère que ces algorithmes peuvent ne pas bien se mettre à l'échelle pour des instances très grandes du MKP. En résumé, bien qu'A* fournisse une solution optimale, cela se fait au prix de temps d'exécution plus longs par rapport à BFS et DFS, surtout pour de plus grandes instances de problème. BFS semble être l'algorithme le plus efficace parmi les trois pour les instances de MKP données, bien que le choix de l'algorithme puisse dépendre des exigences spécifiques du problème, telles que les compromis entre l'optimalité et la vitesse

	Nombre de nœuds visitées			Taux de précision		
Nombre d'objet	DFS	BFS	ASTAR	DFS	BFS	ASTAR
5	6	154	14	100	100	90.76923
6	182	607	46	98.03922	98.03922	88.23529
7	196	1132	52	89.85507	89.85507	85.50725
8	2241	3054	19	77.77778	77.77778	76.76768
9	2051	2966	23	66.26506	66.26506	60.24096
10	4992	4621	57	70.09346	70.09346	64.48598
11	1164	9313	54	57.66423	57.66423	57.66423
12	2821	12501	22	72.8972	72.8972	66.35514
13	13038	19412	8	53.02013	53.02013	53.02013
14	11984	26575	14	49.62406	49.62406	47.36842

Tableau 3: Nombre de nœuds traversé avant d'atteindre le meilleur résultat et le taux de précision.

Pour de petits nombres d'objets (5-7), BFS parcourt significativement plus de nœuds que DFS et A*, mais tous les algorithmes atteignent une précision similaire proche de l'optimal. À mesure que le nombre d'objets augmente de 8 à 10, le nombre de nœuds parcourus par BFS et DFS se rapproche, avec une légère préférence pour DFS à 8 objets et BFS à 9-10 objets. A* maintient une précision proche de BFS et DFS pendant cette période.

Lorsque le nombre d'objets atteint 11, BFS explore de loin le plus grand nombre de nœuds, suivi de DFS, tandis qu'A* parcourt considérablement moins de nœuds. Cependant, les trois algorithmes atteignent la même précision de 57,66% pour cette instance spécifique.

À partir de 12 objets et au-delà, BFS parcourt systématiquement le plus grand nombre de nœuds, suivi de DFS, et A* parcourt le moins de nœuds, ce qui le rend plus évolutif

pour les grandes instances. Pour 12 objets, BFS et DFS ont une précision légèrement supérieure à A*, mais pour 13-14 objets, A* regagne un léger avantage en précision.

Dans l'ensemble, bien que BFS et DFS puissent atteindre une précision légèrement supérieure dans certains cas, leurs performances se dégradent rapidement en termes de nombre de nœuds explorés à mesure que la taille du problème augmente. A*, guidé par sa fonction heuristique, parcourt considérablement moins de nœuds tout en maintenant une précision raisonnablement proche de l'optimal, le rendant plus efficace pour résoudre de plus grandes instances du MKP.

Bien que non capturé dans ces tableaux, il est important de noter que A* peut avoir des frais généraux supplémentaires dus au calcul et à la maintenance de la fonction heuristique, ce qui doit être pris en compte pour une évaluation complète des performances.

5. Conclusion

Ce rapport examine la résolution du problème du sac à dos multiple (MKP) à l'aide de divers algorithmes de recherche exacte tels que la recherche en largeur (BFS), la recherche en profondeur (DFS) et l'algorithme A*. Les expérimentations effectuées comparent les performances de ces algorithmes en termes de nombre de nœuds explorés et de précision de la solution sur des instances de MKP de différentes tailles. Les résultats révèlent que BFS et DFS présentent une précision proche de l'optimal pour de petites instances, mais leur efficacité diminue rapidement avec la taille du problème en raison d'une complexité exponentielle. En revanche, l'algorithme A* maintient un équilibre entre la précision de la solution et le nombre de nœuds explorés grâce à sa fonction heuristique, le rendant plus efficace pour résoudre des instances de grande taille du MKP. Cependant, l'utilisation de l'algorithme A* nécessite la prise en compte de frais généraux supplémentaires liés à la fonction heuristique. En conclusion, bien que les algorithmes BFS, DFS et A* garantissent l'optimalité de la solution, leur applicabilité est limitée pour les instances de MKP de grande taille en raison de leur complexité exponentielle, et des améliorations futures pourraient se concentrer sur l'exploration d'heuristiques plus efficaces pour l'algorithme A* ou sur l'utilisation de méthodes approchées pour obtenir des solutions acceptables dans un délai raisonnable.