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'Electronique et d'informatique
Département d'informatique

Projet Partie 1

Meta heuristique

Recherche basée espace des états
Le problème des Sacs à Dos Multiples (Multiple knapsack problem)

Réalisé par :
OUCHAOU Chafaa
TAOUCI Kenza

Systèmes Informatiques Intelligents

Année scolaire : 2023/2024

Table des matières

CHAPITRE 1
Étude de l'art

1.1 Introduction

L'étude de l'optimisation combinatoire revêt une importance significative, occupant une place prépondérante tant dans le domaine de la recherche opérationnelle que dans celui de l'informatique. En raison de la pertinence de ces problèmes, un grand nombre de techniques de résolution ont été élaborées dans les domaines de la recherche opérationnelle (RO) et de l'intelligence artificielle (IA). Ces approches peuvent être généralement regroupées en deux catégories principales : d'une part, les méthodes exactes (complètes) qui peuvent déterminer la solution optimale si elle existe et démontrer l'insolvabilité du problème dans le cas contraire ; d'autre part, les méthodes approximatives (incomplètes) qui sacrifient la complétude pour gagner en efficacité.

1.2 La théorie de la complexité

La complexité des algorithmes évalue le temps de calcul en fonction de la taille du problème à résoudre, elle est mesurée par le nombre d'opérations élémentaires requises. Selon les objectifs, on peut établir la complexité dans le meilleur des cas, le pire des cas ou en moyenne. Pour le problèmes du sac à dos multiple, étant un problème NP-complet, on développe des méthodes exactes et non exactes pour optimiser les performances et réduire le temps de calcul.

f(n)	**Complexité de l'algorithme**
O(1)	Complexité constante
O(log(n))	Complexité logarithmique
O(n)	Complexité linéaire
O(nlog(n))	Complexité quasi-linéaire
$O(n^2)$	Complexité quadratique
$O(n^3)$	Complexité cubique
$O(n^p)$	Complexité polynomiale
$O(n^p log(n))$	Complexité quasi-polynomiale
$O(2^n)$	Complexité exponentielle
O(n !)	Complexité factorielle

TABLE 1.1 – Les différents types de complexité

1.3 Méthodes de résolution des problèmes

La résolution des problèmes d'optimisation a conduit à l'émergence de diverses méthodes [3]. Traditionnellement, les approches exactes étaient privilégiées, mais avec des problèmes de grande taille, leur efficacité diminue considérablement en raison du temps nécessaire pour explorer l'ensemble des solutions. Ainsi, les méthodes approximatives sont devenues populaires, offrant des solutions quasi optimales dans des délais raisonnables.

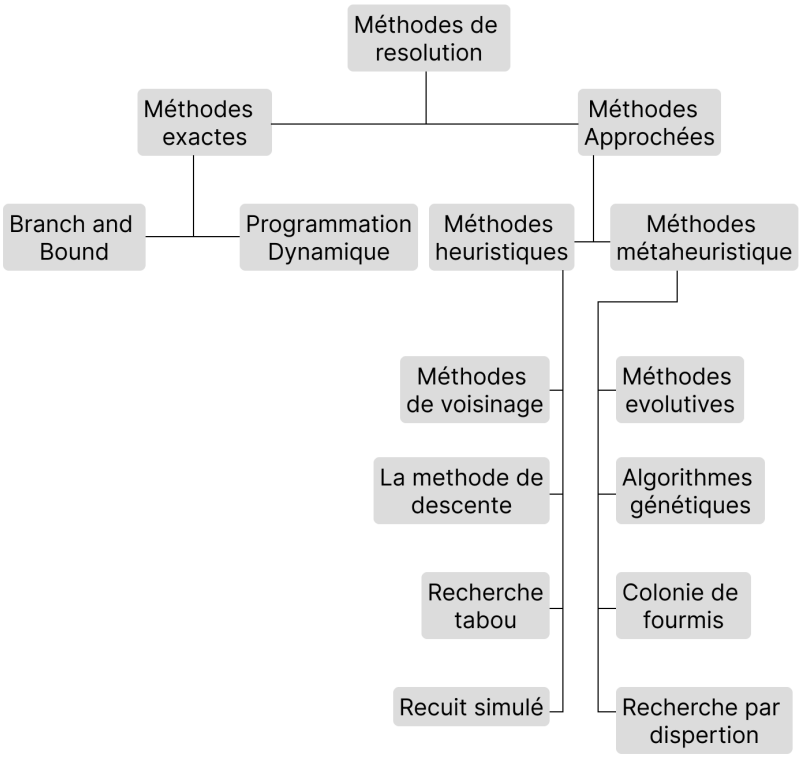

FIGURE 1.1 – Classification des méthodes de résolution

1.4 Méthodes exactes

Le principe des méthodes exactes (Les approches complètes) consiste à rechercher, de manière implicite, une solution, la meilleure solution ou l'ensemble des solutions d'un problème. En général, les méthodes exactes sont utiles dans les cas où le temps de calcul nécessaire pour atteindre la solution optimale n'est pas excessif, ce qui correspond à des problèmes de petite taille. Parmi ces méthodes, on cite :

— la méthode de Séparation et d'Évaluation Progressive (banch and bound),

— la programmation dynamique.

1.4.1 Méthode de Branch-and-Bound

L'algorithme de séparation et évaluation, plus connu sous son appellation anglaise Branch and Bound (B&B), repose sur une méthode arborescente de recherche d'une solution optimale par séparations et évaluations, en représentant les états solutions par un arbre d'états, avec des nœuds et des feuilles. Le branch-and-bound peut se baser sur l'une de ces stratégies de parcours :
La largeur d'abord : Cette stratégie favorise les sommets les plus proches de la racine en effectuant moins de séparations du problème initial.
La profondeur d'abord : Cette stratégie privilégie les sommets les plus éloignés de la racine (de la profondeur la plus élevée) en appliquant plus de séparations au problème initial. Cette voie mène rapidement à une solution optimale en économisant la mémoire.

1.4.2 La programmation dynamique

La programmation dynamique est une méthode couramment utilisée en recherche opérationnelle, adaptée aux problèmes où une solution optimale peut être dérivée des solutions optimales de ses sous-problèmes. Ces sous-problèmes doivent partager la même structure que le problème d'origine et être décomposés de manière récursive. Un exemple classique est le problème de la somme de sous-ensembles, qui peut être résolu efficacement avec un algorithme pseudo-polynomial utilisant la programmation dynamique. Ce problème est un cas particulier du problème du sac à dos

1.5 Méthodes approchées

Il existe deux types de méthodes approchées : les méthodes heuristiques et les méthodes métaheuristiques.

1.5.1 Les heuristiques

En optimisation combinatoire, une heuristique est un algorithme approché qui permet d'identifier en temps polynomial au moins une solution réalisable rapidement, mais pas obligatoirement optimale. L'algorithme A* est une méthode

heuristique. Il utilise une heuristique pour guider la recherche vers une solution potentielle en évaluant les coûts des chemins possibles à chaque étape.

1.5.2 Les méta heuristiques

Les méta heuristiques constituent une classe de méthodes qui fournissent des solutions de bonne qualité en temps raisonnable à des problèmes combinatoires réputés difficiles pour lesquels on ne connaît pas de méthode classique plus efficace. Parmi les méthodes méta heuristiques, on cite l'algorithme génétique qui est une méthode de résolution de problèmes inspirée du processus de sélection naturelle et de génétique. Il est utilisé pour résoudre des problèmes d'optimisation et de recherche, en simulant l'évolution d'une population de solutions candidates au fil des générations.

CHAPITRE 2

Le problème des Sacs à Dos Multiples (Multiple knapsack problem)

2.1 Le problème du sac à dos

Le problème du sac à dos, ou Knapsack Problem (KP) en anglais, est un problème classique d'optimisation combinatoire appartenant à la classe des problèmes NP-complets. Ce problème est parmi les plus étudiés dans le domaine de la recherche opérationnelle. Il peut être formulé comme suit : étant donné un ensemble de n objets, où chaque objet i est caractérisé par un poids wi et un profit pi. L'objectif est de trouver le sous-ensemble d'objets à charger dans un sac de capacité c afin de maximiser la somme des profits.

2.2 Le problème du sac à dos Multiple

Le problème du sac à dos multiple[2], ou Multiple Knapsack Problem (MKP) en anglais, est une généralisation du problème standard du sac à dos, où l'on cherche à remplir m sacs de différentes capacités au lieu de considérer un seul sac. On considère un ensemble N = 1, ..., n d'objets à charger dans m sacs à dos de capacités ci avec i dans 1, ..., m. Chaque objet j dans N est caractérisé par son poids wj et son profit pj. Il s'agit alors de trouver m sous-ensembles disjoints de N (où chaque sous-ensemble correspond au remplissage d'un sac) qui maximisent le profit total formé par la somme des objets sélectionnés.

2.3 Exemple

Par exemple on suppose qu'on a le problème MKP avec 4 items, 2 sacs : capacité des sac ci = (10,7). Les profits et les poids des items (wj ,pj) = j1(9,3), j2(7,3), j3(6,7), j4(1,5). Alors, la solution optimale à cet exemple de MKP est d'affecter

l'item j1(9,3) et j4(1,5) au sac de capacité 10, et l'item j3(6,7) au sac de capacité 7.

2.4 Exemples d'application

Le MKP compte plusieurs applications industrielles dont on peut citer quelques exemples :

- Le chargement de fret sur les navires :il s'agit de choisir certains conteneurs, dans un ensemble de n conteneurs à charger dans m navires de différentes capacités de chargement.
- Le chargement de n réservoirs par m liquides.
- La sélection d'investissement (Capital Budgeting Problem) de manière à maximiser le rendement, sans bien sûr, dépasser la somme disponible ;

2.5 Résolution du problème des sacs à dos multiples

Dans cette section, nous allons résoudre le problème du sac à dos multiple par plusieurs manières[1], en commençant par les méthodes exactes puis les méthodes heuristiques. Pour chaque approche, nous parlerons du fonctionnement de l'algorithme et nous expérimenterons sur la taille du problème.

2.5.1 Approche en largeur d'abord BFS

Dans cet algorithme, nous utilisons une file d'attente pour explorer les états possibles. Au début, nous ajoutons le nœud initial à la file d'attente. Ensuite, nous retirons un nœud de la file d'attente (ensemble ouvert) à chaque itération et examinons son état actuel. Si la profondeur de ce nœud atteint la limite fixée, nous évaluons la solution pour ce chemin. Sinon, pour chaque sac disponible, nous générons un nouvel état en ajoutant l'élément suivant à ce sac et l'ajoutons à la file d'attente pour une exploration ultérieure. Nous répétons ce processus jusqu'à ce que la file d'attente soit vide, explorant ainsi tous les chemins possibles.

2.5.2 Algorithme

Algorithme 1 : BFS(startNode, limit, sacs, items)

Input : startNode, limit, sacs, items

Output : Meilleure solution trouvée

```
1  Créer une file d'attente queue;
2  Ajouter startNode à queue;
3  while queue n'est pas vide do
4      Extraire le nœud currentNode de queue;
5      Obtenir le chemin actuel currentPath de currentNode;
6      if La profondeur de currentNode est égale à limit then
7          Vérifier la solution pour currentPath et mettre à jour la meilleure
            solution si nécessaire;
8      end
9      if La profondeur de currentNode est inférieure à limit then
10         foreach sac dans sacs do
11             Créer un nouveau nœud node en ajoutant l'objet suivant à
                currentNode pour chaque sac;
12             Ajouter node à la fin du chemin actuel currentPath;
13             Ajouter currentPath à node;
14             Ajouter node à queue;
15         end
16     end
17 end
18 Retourner la meilleure solution trouvée;
```

2.5.3 Exemple

Pour illustrer l'exécution de l'algorithme BFS, nous allons le dérouler et dessiner l'arbre pour un exemple contenant 2 sacs de capacités 13 et 25 et 3 items Ji(wi,vi) tels que J1(3,73), J2(20,55), J3(13,61). L'algorithme parcourt les nœuds et chaque fois qu'il arrive à un nœud feuille, il vérifie les contraintes de satisfaisabilité. Si le poids d'un sac est dépassé, il retourne -1 ; sinon, il retourne la somme

des valeurs des objets.

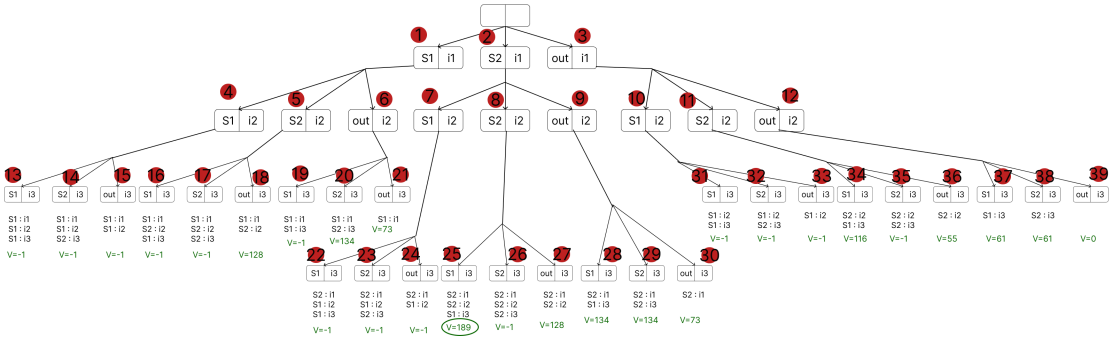

FIGURE 2.1 – L'arbre montrant le déroulement de l'algorithme BFS.

2.5.4 Experimention

Dans cette section, nous allons calculer le temps d'exécution en millisecondes de l'algorithme BFS en fonction du nombre de sacs et d'objets. Nous ferons varier ces deux paramètres et calculerons le temps. L'analyse des résultats sera effectuée ultérieurement dans ce rapport en combinant les résultats des 3 algorithmes.

Remarque : la notation "X" dans la table signifie que l'algorithme n'a pas pu résoudre le problème en raison du nombre important de nœuds à développer, pour l'algorithme BFS Il s'agit généralement d'un problème de mémoire.

9	0	0	0	5	37	407	X	X	X
8	0	0	0	5	90	460	5255	X	X
7	0	0	0	1	10	113	2652	X	X
6	0	0	0	0	5	53	900	7195	X
5	0	0	0	0	2	25	95	2414	13506
4	0	0	0	0	0	4	27	298	2835
3	0	0	0	0	0	1	5	20	150
2	0	0	0	0	0	0	2	5	7
1	1	0	0	0	0	0	0	0	1
Sacs/Item	1	2	3	4	5	6	7	8	9

TABLE 2.1 – Temps d'exécution en millisecondes en fonction du nombre de sacs et d'objets "BFS".

2.5.5 Approche en profondeur d'abord DFS

Dans cet algorithme, nous utilisons une pile pour explorer les états possibles. Nous commençons par ajouter le nœud initial à la pile. Ensuite, nous retirons un nœud de la pile à chaque itération et examinons son état actuel. Si la profondeur de ce nœud atteint la limite fixée, nous évaluons la solution pour ce chemin. Sinon, pour chaque sac disponible, nous générons un nouvel état en ajoutant l'élément suivant à ce sac et l'ajoutons à la pile pour une exploration ultérieure. Nous répétons ce processus jusqu'à ce que la pile soit vide, explorant ainsi tous les chemins possibles. Un paramètre peut être spécifié pour déterminer la limite de profondeur en cas de branche trop longue.

2.5.6 Algorithme

Algorithme 2 : DFS(startNode, limit, sacs, items)

Input : startNode, limit, sacs, items
Output : Meilleure solution trouvée

```
1  Créer une pile stack;
2  Ajouter startNode à stack;
3  while stack n'est pas vide do
4      Extraire le nœud currentNode de stack;
5      Obtenir le chemin actuel currentPath de currentNode;
6      if La profondeur de currentNode est égale à limit then
7          Vérifier la solution pour currentPath et mettre à jour la meilleure
           solution si nécessaire;
8      end
9      if La profondeur de currentNode est inférieure à limit then
10         foreach sac dans sacs do
11             Créer un nouveau nœud node en ajoutant l'objet suivant à
               currentNode pour chaque sac;
12             Ajouter node à la fin du chemin actuel currentPath;
13             Ajouter currentPath à node;
14             Ajouter node à stack;
15         end
16     end
17 end
18 Retourner la meilleure solution trouvée;
```

2.5.7 Exemple

Pour illustrer l'exécution de l'algorithme DFS, nous allons le dérouler et dessiner l'arbre pour un exemple contenant 2 sacs de capacités 13 et 25 et 3 items Ji(wi,vi) tels que J1(3,73), J2(20,55), J3(13,61). L'algorithme parcourt les nœuds et chaque fois qu'il arrive à un nœud feuille, il vérifie les contraintes de satisfaisabilité. Si le poids d'un sac est dépassé, il retourne -1 ; sinon, il retourne la somme

des valeurs des objets.

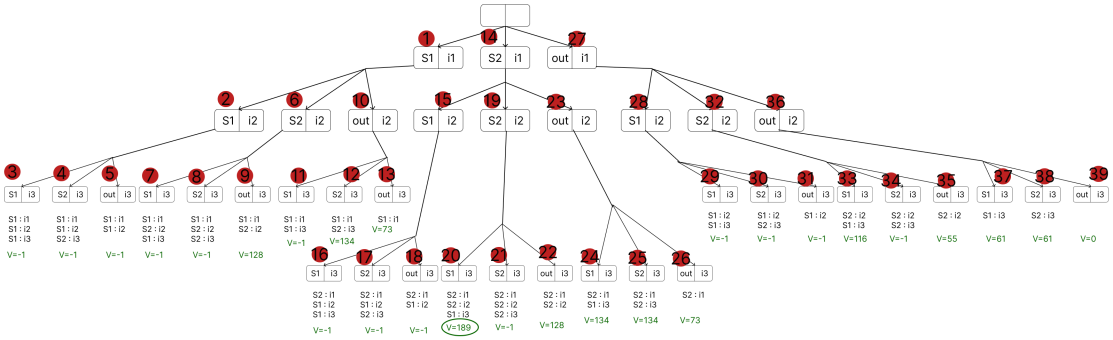

FIGURE 2.2 – L'arbre montrant le déroulement de l'algorithme DFS.

La Figure 2.2 montre un exemple de déroulement de l'algorithme DFS sur l'instance énoncé précédemment.

2.5.8 Experimention

Dans cette section, nous allons calculer le temps d'exécution en millisecondes de l'algorithme DFS en fonction du nombre de sacs et d'objets. Nous ferons varier ces deux paramètres et calculerons le temps. L'analyse des résultats sera effectuée ultérieurement dans ce rapport en combinant les résultats des 3 algorithmes.

Remarque : la notation "X" dans la table signifie que l'algorithme n'a pas pu résoudre le problème en raison du nombre important de nœuds à développer pour l'algorithme DFS. Il s'agit généralement d'un problème de calcule qui prend trop de temps.

9	0	0	0	3	37	315	3559	40956	X
8	0	0	0	4	34	182	1473	16505	131553
7	0	0	0	1	10	94	810	6043	58554
6	0	0	0	0	4	35	410	1925	14693
5	0	0	0	0	3	19	99	626	4299
4	0	0	0	0	0	4	29	210	877
3	0	0	0	0	0	1	4	20	100
2	0	0	0	0	0	0	0	2	7
1	1	0	0	0	0	0	0	0	1
Sacs/Item	1	2	3	4	5	6	7	8	9

TABLE 2.2 – Temps d'exécution en millisecondes en fonction du nombre de sacs et d'objets "DFS".

2.5.9 Approche algorithme A étoile

L'algorithme A* est une méthode de recherche guidée par une fonction objectif qui combine le coût du chemin parcouru jusqu'à présent (g) avec une estimation du coût restant pour atteindre l'objectif (h). Il utilise une fonction d'évaluation pour évaluer les nœuds à explorer, en prenant en compte la valeur (g + h). A* explore les nœuds en fonction de cette fonction d'évaluation, en priorisant les nœuds avec le coût total le plus faible (minimisation) ou bien les coût total le plus élevé (maximisation).

La fonction objectif

Pour la résolution du problème nous avons choisi la fonction objectif suivant :

$$F = G + H$$

Où :

$$G$$

est la somme des poids des items depuis le début jusqu'au nœud courant.

$$H$$

est la différence entre la taille du sac et la taille de l'item à mettre dans le sac.

2.5.10 Algorithme

Algorithme 3 : AStarSolver(startNode, limit, sacs, items)

Input : startNode, limit, sacs, items

Output : Meilleure solution trouvée

```
1  Créer une file de priorité queue avec une fonction de comparaison basée
     sur g + h;
2  Ajouter startNode à queue;
3  while queue n'est pas vide do
4      Extraire le nœud currentNode de queue;
5      Obtenir le chemin actuel currentPath de currentNode;
6      if La profondeur de currentNode est égale à limit then
7          Vérifier la solution pour currentPath;
8          Mettre à jour la meilleure solution si nécessaire et retourner;
9      end
10     if La profondeur de currentNode est inférieure à limit then
11         foreach sac dans sacs do
12             Créer un nouveau nœud node en ajoutant l'élément suivant à
                 currentNode pour chaque sac;
13             Calculer g et h pour node;
14             Ajouter node à la fin du chemin actuel currentPath;
15             Ajouter currentPath à node;
16             Ajouter node à queue;
17         end
18     end
19 end
20 Retourner la meilleure solution trouvée;
```

2.5.11 Exemple

Pour illustrer l'exécution de l'algorithme A étoile, nous allons le dérouler et dessiner l'arbre pour un exemple contenant 2 sacs de capacités 13 et 25 et 3 items Ji(wi,vi) tels que J1(3,73), J2(20,55), J3(13,61).

L'algorithme parcourt les nœuds et quand il arrive à un nœud feuille il vérifie les contraintes de satisfaisabilité. Si le poids d'un sac est dépassé, il retourne -1 ; sinon,il retourne la somme des valeurs des objets.

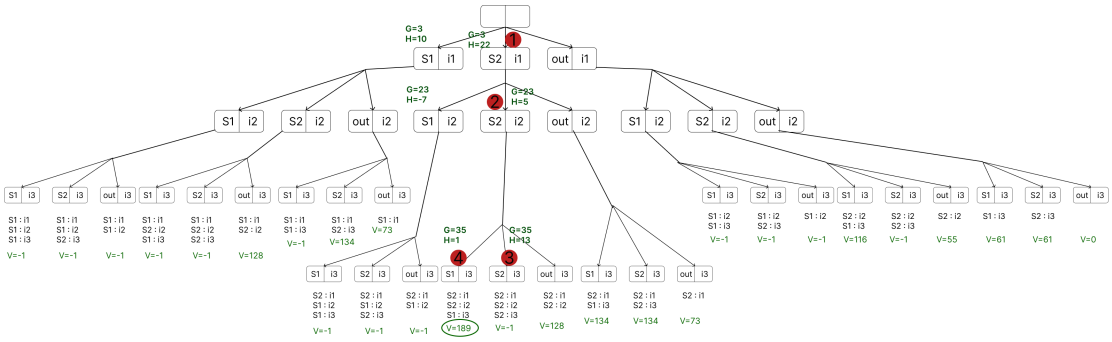

FIGURE 2.3 – L'arbre montrant le déroulement de l'algorithme 1 étoile.

2.5.12 Experimention

Dans cette section, nous allons calculer le temps d'exécution en millisecondes de l'algorithme A étoile en fonction du nombre de sacs et d'objets. Nous ferons varier ces deux paramètres et calculerons le temps. L'analyse des résultats sera effectuée ultérieurement dans ce rapport en combinant les résultats des 3 algorithmes.

9	0	0	0	4	1	17	169	1928	11178
8	0	0	0	5	18	16	125	1141	4062
7	0	0	0	1	14	113	990	160	1612
6	0	0	0	0	6	48	374	142	528
5	0	0	0	0	2	20	127	28	206
4	0	0	0	0	1	0	37	39	215
3	0	0	0	0	0	0	0	3	10
2	0	0	0	0	0	0	0	0	2
1	0	0	0	0	0	0	0	0	0
Sacs/Item	1	2	3	4	5	6	7	8	9

TABLE 2.3 – Temps d'exécution en millisecondes en fonction du nombre de sacs et d'objets "A étoile".

2.6 L'analyse et la comparaison

Avant de commencer l'analyse, il est important de spécifier que les tests ont été effectués sur une machine équipée d'un processeur Intel Core i5 de 4ème génération et de 8 Go de RAM.

Après l'exécution avec différentes valeurs de sacs et d'objets, nous constatons que le temps d'exécution des trois algorithmes suit le même modèle. Ce modèle se compose de trois parties :

1 : Une partie où le nombre de nœuds à développer est inférieur à 100 000. Dans cette partie, le temps d'exécution se rapproche considérablement entre les trois algorithmes, à quelques millisecondes près. Parfois, on remarque que l'algorithme A* est plus long que les deux autres, cela est dû aux opérations d'ordonnancement des nœuds qui consomment du temps sans apporter de bénéfices significatifs. En règle générale, l'algorithme A* est le plus rapide, suivi de l'algorithme DFS, et enfin de l'algorithme BFS.
2 : La partie où le nombre de nœuds à développer est supérieur à 100 000 mais inférieur à 15 millions. Dans cette partie, on remarque une différence significative de temps d'exécution entre l'algorithme A* et les deux autres algorithmes BFS et DFS. Cela est dû au fait que l'algorithme A* ne développe pas tous les nœuds pour arriver à la solution. Le temps d'exécution de cet algorithme est généralement 20 fois inférieur aux deux autres. On constate aussi que, en général, l'algorithme DFS est plus rapide que l'algorithme BFS.
3 : La partie où le nombre de nœuds à développer est compris entre 15 millions et 1 000 000 000. Dans cette partie, l'algorithme BFS est incapable d'arriver à une solution en raison du nombre important de nœuds, tandis que l'algorithme DFS est capable d'arriver à des solutions et de résoudre le problème, mais n'a pas pu résoudre le problème avec 1 000 000 000 de nœuds à développer. En revanche, l'algorithme A* se comporte bien et fournit des résultats satisfaisants en un temps raisonnable, même lorsque le nombre de nœuds à développer dépasse 1 000 000 000. puisqu'il ne visite pas toutes les solutions possibles, ce qui lui permet de résoudre le problème en peu de temps.

2.6.1 graphe de comparaison

Ci-dessous, nous allons fournir à la fois un tableau et un graphique illustrant les variations de temps d'exécution entre les trois algorithmes. Et pour cella nous allons sommer les temps d'execution des différentes combinaisons d'objets dans un sac i.

A star	0	2	13	292	383	1098	2890	5367	13297
DFS	1	9	125	1120	5046	17067	65512	149751	x
BFS	1	14	176	3146	16042	X	X	X	X
Configuration	1	2	3	4	5	6	7	8	9

TABLE 2.4 – La somme des temps d'exécution des configurations

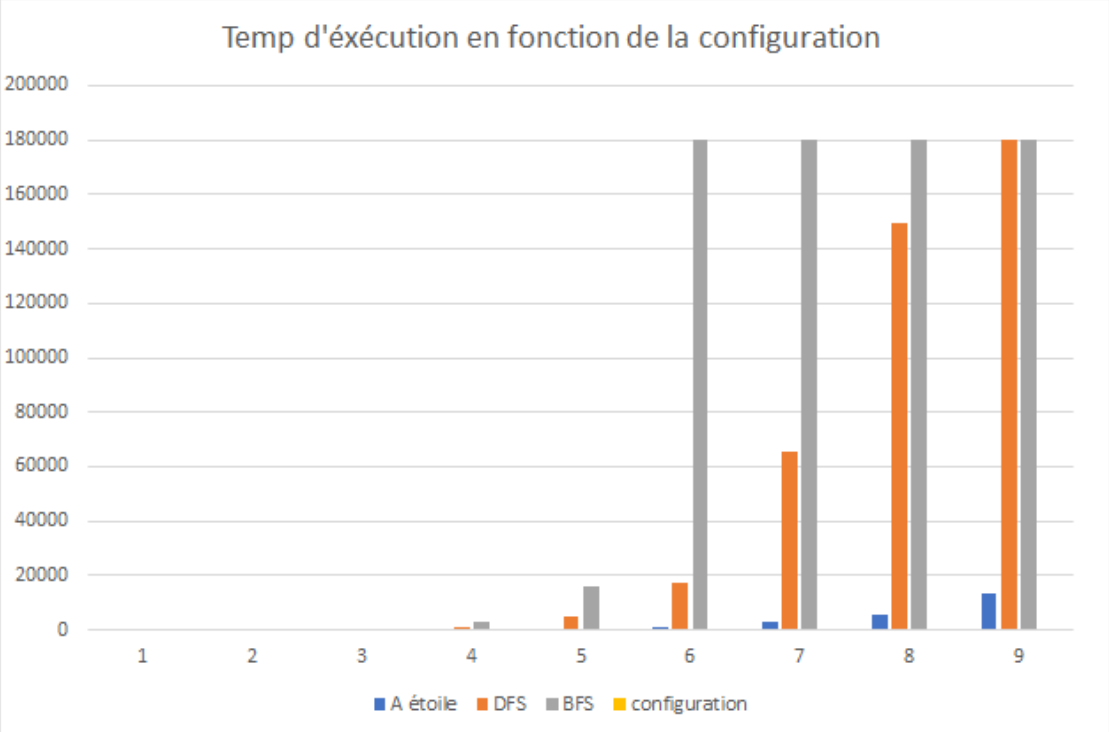

FIGURE 2.4 – Temps d'exécution en milisecondes de chaque algorithme

Remarque : Dans le graphe, on s'arrête à 180 000 millisecondes pour bien visualiser la différence de temps entre les différents algorithmes.

Comme on peut le voir sur la figure, on peut remarquer que l'algorithme A*

est plus rapide par rapport aux deux autres.

La croissance en temps d'exécution est trop rapide pour l'algorithme BFS et DFS et cela pour des nombres de sacs et d'items petits.

2.7 Conclution

En conclusion, cette premiere partie du projet a traiter la resolution du probleme du sac a dos multiple avec une aproche exect BFS et BFS et par l'utilisation de l'algoritheme A etoile qui utlise une heuristique pour gider la recherche

l'analyse des temps d'exécution des trois algorithmes, à savoir l'algorithme A*, BFS et DFS, révèle des tendances significatives. Dans des situations où le nombre de nœuds à développer est relativement faible, les trois algorithmes présentent des performances comparables, avec des temps d'exécution proches. Cependant, lorsque le nombre de nœuds à développer augmente, des différences substantielles apparaissent.

L'algorithme A* se distingue généralement par son temps d'exécution inférieur, grâce à sa capacité à utiliser une heuristique pour guider la recherche vers la solution optimale tout en évitant de développer tous les nœuds. En revanche, les algorithmes BFS et DFS, bien qu'ils puissent être efficaces dans certains cas, montrent une dégradation significative des performances lorsque le nombre de nœuds devient très important.

Bibliographie

[1] geeksforgeeks, 2024.

[2] David Pisinger. An exact algorithm for large multiple knapsack problems. *European Journal of Operational Research*, 114(3) :528–541, 1999.

[3] Mr BALBAL Samir. Utilisation de l'intelligence artificielle pour résoudre le problème du sac à dos, : 2014-2015.