

Projet Métaheuristique-partie1-

Problème de sac à dos multiple

Recherche basée espace des états

Binômes :

Mokrani ikram, mat :202031037505

Louaifi azouaou, mat :202031044812

Enseignante :

Messouadi imene

06/04/2024

Introduction

Le problème du sac à dos multiple, ou Multiple Knapsack Problem (MKP) en anglais, est une généralisation du problème standard du sac à dos, où on essaie de remplir m sacs de différentes capacités au lieu de considérer un seul sac et cela sous la contrainte de respecter la capacité de ces derniers et dont l'objectif est de maximiser le profit global. Les problèmes de sac-à-dos sont en général NP-Complet et par conséquent la complexité en temps est exponentielle.

Instance :

On considère un ensemble $N = \{1, \dots, m\}$ d'items à charger dans m sacs à dos de capacité c_i avec $i \in \{1, \dots, m\}$. Chaque item $j \in N$ est caractérisé par son poids w_j , son profit p_j . Pour chaque sac à dos i , un ensemble d'objets est présent, représentés par un vecteur où chaque élément vaut 1 si item j est chargé dans le sac i et 0 dans le cas contraire.

Problématique :

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 items sélectionnés.

Exemple :

Considérons un exemple concret pour illustrer le problème du sac à dos multiple :

Supposons que nous disposons de cinq objets avec les caractéristiques suivantes :

objets	1	2	3	4	5
poids	450	580	150	200	50
valeurs	140	420	500	322	190

Nous disposons de trois sacs à dos, chacun ayant une capacité spécifique :

Sac 1 : Capacité de 700

Sac 2 : Capacité de 520

Sac 3 : Capacité de 200

Sacs :

Capacité des sacs	700	520	200
-------------------	-----	-----	-----

Validation de la solution :

Pour chaque sac à dos, on doit s'assurer que la somme des poids des objets qu'il contient ne dépasse pas sa capacité maximale.

On doit s'assurer que chaque objet est inclus dans exactement un sac à dos.

Evaluation de la solution :

L'évaluation d'une solution se fait lorsque tous les objets sont placés dans les sacs ou lorsque les capacités restantes des sacs ne permettent pas l'ajout d'un autre objet.

À ce stade, la valeur totale des objets placés dans les sacs est calculée qui est dans le but de la maximiser, et cette valeur est retenue pour être affichée.

Représentation de la solution :

Une solution peut être représentée de deux manières :

-En utilisant une liste de sacs à dos, où chaque sac contient une sélection d'objets.

Pour 3 sacs et un objet une solution peut être la suivante :

((obj1),(obj3),(obj5)).

-En utilisant un vecteur de sacs où chaque élément du vecteur contient un vecteur binaire d'objets (0 ou 1). La valeur 0 indique que l'objet à cette position n'existe pas dans le sac, tandis que la valeur 1 indique que l'objet existe.

Pour 3 sacs et 5 objets une solution peut être la suivante :

0	1	0	0	0
0	0	1	0	0
1	0	0	0	1

Espace de recherche :

L'espace de recherche dans le problème des sacs à dos multiples consiste en toutes les combinaisons possibles d'objets à placer dans les différents sacs, tout en respectant les contraintes de capacité de chaque sac.

***Etat initiale** : liste vide où les sacs ne contiennent aucun objet $(((),(),()))$ ou bien vecteur de sacs où chaque sac contient un vecteur d'objets initialisé à 0 pour dire qu'aucun objet n'est présent.

***Etat finale** : qui peut être une solution ou plusieurs solutions valides à notre problème, par ex $((obj1,obj2,obj3),(obj4),())$.

***Les opérateurs à appliquer** :

-Ajouter ou supprimer un objet d'un sac à dos, permettant ainsi d'explorer différentes combinaisons d'objets dans les sacs. (Supprimer veut dire retourner dans l'arborescence).

-Déplacer un objet d'un sac à dos à un autre, ce qui peut aider à optimiser la distribution des objets et d'explorer l'espace de recherche encore et encore et permet aussi d'équilibrer les charges entre les sacs.

-Permuter l'ordre des sacs à dos dans la solution, ce qui peut avoir un impact sur la distribution des objets et sur l'utilisation des capacités des sacs.

*Puis vient l'analyse des chemins du graphe de l'état initiale vers l'état finale pour évaluer les solutions et choisir la solution la plus optimale.

* Une solution est définie par la séquences d'actions appliquer à l'état initial pour arriver à un état final.

Résolution du problème :

Les approches proposées dans la littérature, pour résoudre les problèmes de la famille du sac à dos sont soit des méthodes exactes soit heuristiques. Les méthodes exactes sont capables de résoudre un problème à l'optimalité mais dans un temps exponentiel. Les méthodes heuristiques fournissent une solution approchée, de bonne qualité, dans des laps de temps raisonnables.

Méthodes exactes :

La stratégie de parcours en largeur d'abord : Cette stratégie favorise les sommets les plus proches de la racine en faisant moins de séparations du problème initial.

La stratégie de parcours en profondeur d'abord : Cette stratégie avantage les sommets les plus éloignés de la racine (de profondeur la plus élevée) en appliquant plus de séparations au problème initial. Cette voie mène rapidement à une solution optimale en économisant la mémoire.

Méthodes heuristiques :

Heuristique En optimisation combinatoire, une heuristique est un algorithme approché qui permet d'identifier en temps polynomial au moins une solution réalisable rapide, pas obligatoirement optimale. L'usage d'une heuristique est efficace pour calculer une solution approchée d'un problème et ainsi accélérer le processus de résolution exacte. Généralement une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre sans offrir aucune garantie quant à la qualité de la solution calculée.

Modélisation du problème :

On a fait 2 modélisations pour chaque approche on vous les présente juste en dessous :

1ere modélisation :

Les objets : Un objet est caractérisé par son poids et sa valeur.

Pour représenter les n objets on a opté pour un vecteur d'objets tel que chaque $v[i]$ est un objet.

(poids, valeur)	(poids, valeur)	(poids, valeur)	(poids, valeur)	(poids, valeur)	(poids, valeur)
-----------------	-----------------	-----------------	-----------------	-----------------	-----------------

Les sacs à dos : Un sac à dos est caractérisé par :

-Sa capacité maximale $poids_{max}$.

-Son poids courant présent dans le sac initialement 0.

-sa Valeur courante qui présente la valeur des objets qu'on a dans le sac initialement 0

-Tableau binaire (0 ou 1) de taille égale au nombre d'objets existants tel que $tab[i]=0$ si l'objet[i] est dans le sac, initialement tous les éléments du tab sont à 0.

Pour représenter les n sacs : on va créer un tableau de type sac à dos comme suit :

Poidsmax=700	Poidsmax=520	Poidsmax=200
--------------	--------------	--------------

Poidscour=0 Valcour=0	Poidscour=0 Valcour=0	Poidscour=0 Valcour=0
0 0 0 0 0	0 0 0 0 0	0 0 0 0 0

2ème modélisation :

Les sacs à dos :

On a déclaré une arraylist de taille égale au nombre de sacs et dont chaque élément est une arraylist d'objets .

➔ Pour notre exemple on a 5 sacs donc la taille de l'arraylist est de 5.

((liste d'objets),(liste d'objets),(liste d'objets),(liste d'objets),(liste d'objets)).

Pseudo code des deux modélisations :

La première :

```
public class pile {
    protected Sacados [] sac ;
    protected int niv;
    public pile(int niv,Sacados sac [] ) {
        this.sac = sac;
        this.niv = niv;
    }
}
```

La deuxième :

```
public class Pile2 {
    int[] poidscour ;
    int f ;
    ArrayList<ArrayList<objet>> sacs = new ArrayList<>();

    public Pile2(int f, ArrayList<ArrayList<objet>> listObjet,int poidscour[]) {

        this.f = f;
        this.sacs = listObjet;
        this.poidscour=poidscour;
    }
}
```

Stratégie de recherche en profondeur d'abord (dfs) :

Pseudo code :

Algorithme dfs :

Entrée : objet [] obj, Sacàdos [] sacs(état initiale).

Ouvert : pile de nœuds initialement vide.

Ferme :file de de nœuds initialement vide.

Sortie : une solution si succès sinon échec.

Debut

```
startTime = currentTime()
niv_cour = 0
solcourante = 0
nbr = 0
ferme = Nouvelle liste de tableaux de Sacàdos
sol = Nouveau tableau de Sacàdos de taille taille(sacs)
ouvert = Nouvelle file de type Deque
```

```
// Initialisation de la pile ouverte avec l'état initial
element_pile = Nouvelle pile(niv_cour, sacs)
ouvert.ajouter(element_pile)
```

```
TANT QUE (la pile ouverte n'est pas vide)
  // Retirer l'élément du sommet de la pile ouverte
  element_pile = ouvert.retirer()
```

```
SI (la taille de ferme est égale à la taille de obj OU nbfaux est égal à nbfiles)
```

```
  nbfaux = 0
```

```
  SI (element_pile.niv est égal à niv_cour)
```

```
    ferme.retirerDernierElement()
```

```
  SINON
```

```
    POUR i DE 0 À niv_cour - element_pile.niv FAIRE
```

```
      SI (ferme a plus de 1 élément)
```

```
        ferme.retirerDernierElement()
```

```
      FIN SI
```

```
    FIN POUR
```

```
    niv_cour = element_pile.niv
```

```
  FIN SI
```

```
FIN SI
```

```
cpe = 0
```

```
SI (la taille de ferme n'est pas égale à 0)
```

```
  POUR CHAQUE sac DANS sacs FAIRE
```

```
    SI (le poids courant de sac est supérieur au poids maximal de sac)
```

```
      cpe++
```

```
    FIN SI
```

```
  FIN POUR
```

```
FIN SI
```

```

SI (cpe est supérieur ou égal à 1)
    nbfaux++
SINON
    nbr++
    ferme.ajouter(element_pile.sacs)

SI (la taille de ferme est inférieure à la taille de obj)
    niv_cour++
    nbfiles = 0
    POUR i DE 0 À la taille de obj FAIRE
        a = vrai
        POUR CHAQUE tableau DANS ferme FAIRE
            cpt = 0
            POUR CHAQUE sac DANS sacs FAIRE
                SI (le tableau[sac][i] n'est pas égal à 1)
                    cpt++
            FIN SI
        FIN POUR
        SI (cpt n'est pas égal à la taille de sacs)
            a = faux
            QUITTER LA BOUCLE INTERNE
        FIN SI
    FIN POUR
    SI (a est vrai)
        POUR CHAQUE sac DANS sacs FAIRE
            nbfiles++
            sac1 = Nouveau tableau de sacs de taille taille(sacs)

            // Copier les valeurs actuelles des sacs dans sac1
            POUR CHAQUE p DE 0 À taille(sacs) FAIRE
                sac1[p] = Nouveau Sacàdos(avec les mêmes valeurs que sacsCourants[p])
            // Copier les valeurs actuelles des sacs dans sac1
            POUR CHAQUE p DE 0 À taille(sacs) FAIRE
                sac1[p] = Nouveau Sacàdos(avec les mêmes valeurs que sacsCourants[p])

            sac1[j].tab[i] = 1
            sac1[j].poidcour += obj[i].poids
            sac1[j].valcour += obj[i].valeur
            element = Nouvelle pile(niv_cour, sac1)
            ouvert.ajouter(element)
        FIN POUR
    FIN SI
    FIN POUR
    FIN SI
FIN SI

SI (la taille de ferme est égale à la taille de obj OU nbfaux est égal à nbfiles)
    SI (nbfaux est égal à nbfiles)
        niv_cour--
    FIN SI

```

```

y = 0
POUR CHAQUE sac DANS le dernier élément de ferme FAIRE
    y += sac.valcour
FIN POUR

SI (solcourante < y)
    sol = Copie du dernier élément de ferme
    solcourante = y
FIN SI
FIN SI
FIN TANT QUE

endTime = currentTime()
duration = endTime - startTime
tempsEcouleEnSecondes = duration / 1_000_000_000.0
AFFICHER "Temps écoulé en secondes : " + tempsEcouleEnSecondes
AFFICHER "nbrnoeud : " + nbr
AFFICHER "La valeurtotale : " + solcourante + " Le sac : " + sol
FIN

```

Fonctionnement :

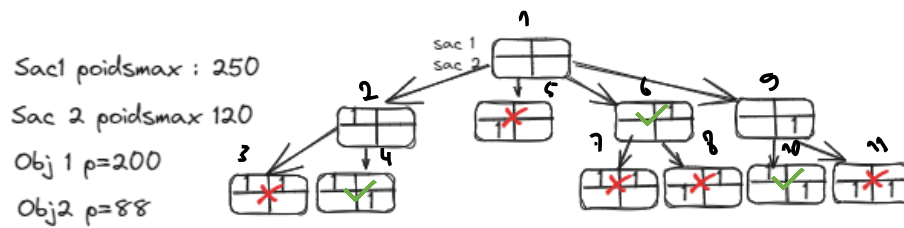
Initialement les sacs sont vides j'empile l'élément « sacs vides » dans ouvert et je fais boucle tant que ouvert n'est pas vide, on dépile l'élément du ouvert après comme c'est le premier élément »ferme est vide » on va directement empiler ses fils dans ouvert .

On reboucle et on dépile un élément du ouvert on va tester si cet élément respecte les contraintes de capacité si oui on l'empile dans ferme et on développe ses fils sinon on incrémente la variable nombre faux qui désigne nombre de fils faux de l'élément récemment empilé dans ferme. Si tous les fils sont faux alors on va évaluer l'élément récemment empilé dans ferme et qui présente une solution qu'on lui calcule la valeurtotale qu'on doit comparer avec la valeurtotale précédente pour prendre la meilleure sinon on va évaluer une solution quand la taille de ferme -1 (on soustrait le premier élément qui représente les sacs vide) soit égale au nombre d'objets.

Quand on évalue une solution, en remontant il y'a un test pour savoir l'élément prochain du ouvert est dans quel niveau et on le compare avec le niveau courant pour retirer les éléments du ferme et explorer d'autres arborescences.

Exemple de fonctionnement :

« Pour 2sacs et 2 objets »



Algorithme Bfs :

Pseudo code :

Algorithme bfs :

Entrée : objet [] obj, Sacàdos [] sacs(état initiale).

Ouvert : file de nœuds initialement vide.

Ferme :file de de nœuds initialement vide.

Sortie : une solution si succès sinon échec.

Debut

startTime = currentNanoTime()

niv_cour = 0

solcourante = 0

nbr = 0

ferme = new ArrayList<Sacàdos[]>

sol = new Sacàdos[sacs.length]

ouvert = new LinkedList<pile>

element_pile = new pile(niv_cour, sacs)

ouvert.add(element_pile)

nbfaux = 0

nbfiles = 0

while ouvert non vide:

element_pile = ouvert.poll()

si taille de ferme - 1 == longueur de obj ou nbfaux == nbfiles:

nbfaux = 0

si element_pile.niv == niv_cour:

ferme.remove(taille de ferme - 1)

sinon:

pour i allant de 0 à niv_cour - element_pile.niv +2:

si taille de ferme >= 2:

ferme.remove(taille de ferme - 1)

niv_cour = element_pile.niv

cpe = 0

si taille de ferme différent de 0:

pour e allant de 0 à longueur de sacs:

si element_pile.sac[e].poidcour > element_pile.sac[e].poidsmax:

cpe++

```

si cpe >= 1:
    nbfaux++
sinon:
    nbr++
    ferme.add(element_pile.sac)
    si taille de ferme - 1 < longueur de obj:
        niv_cour++
        nbfiles = 0
        pour i allant de 0 à longueur de obj:
            k = 0
            a = true
            tant que k < taille de ferme et a est vrai:
                cpt = 0
                pour e allant de 0 à longueur de sacs:
                    si ferme.get(k)[e].tab[i] != 1:
                        cpt++
                si cpt != longueur de sacs:
                    a = faux
                    k++

            si a est vrai:
                pour j allant de 0 à longueur de sacs:
                    nbfiles++
                    sac = nouveau Sacàdos[sacs.length]
                    sacsCourants = ferme.get(taille de ferme - 1)
                    pour p allant de 0 à longueur de sacsCourants:
                        sac[p] = nouveau Sacàdos(sacsCourants[p].poidsmax, sacsCourants[p].poidcour,
sacsCourants[p].valcour, copie de sacsCourants[p].tab)
                    sac[j].tab[i] = 1
                    sac[j].poidcour += obj[i].poids
                    sac[j].valcour += obj[i].valeur
                    element = nouveau pile(niv_cour, sac)
                    ouvert.add(element)

        si taille de ferme - 1 == longueur de obj ou nbfaux == nbfiles:
            si nbfaux == nbfiles:
                niv_cour--
            y = 0
            pour b allant de 0 à longueur de sacs:
                y += ferme.get(taille de ferme - 1)[b].valcour
            si solcourante < y:
                sol = copie de ferme.get(taille de ferme - 1)
                solcourante = y

endTime = currentNanoTime()
duration = endTime - startTime
tempsEcouleEnSecondes = duration / 1_000_000_000.0
afficher "Temps écoulé en secondes : " + tempsEcouleEnSecondes
afficher "nbrnoeud : " + nbr
afficher "la valeur totale : " + solcourante + " le sac : "
FIN .

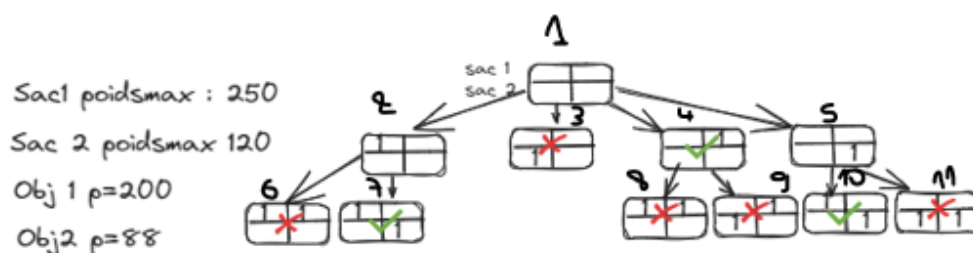
```

Fonctionnement de l'algorithme :

Le fonctionnement est le même que DFS juste la seule différence que dans DFS on a travaillé avec une pile et dans BFS on a travaillé avec une file.

Exemple du fonctionnement :

« Pour 2sacs et 2 objets »



Complexité des deux Algorithmes bfs et dfs :

La complexité est le nombre de nœuds dans le graphe car dfs et bfs parcourent tout le graphe pour tirer la meilleure solution et donc on peut calculer la complexité comme suit :

$$\sum_{n=0}^{nbobj} nbsac^{n+1} \frac{nbobj!}{nbobj - n - 1}$$

La complexité c'est nb sac puissance nbobj = nbsac puissance nb objets

Algorithme A* :

Au pire des cas, l'arborescence est construite entièrement (cas d'échec).

Sinon il va dépendre de la fonction f(n).

L'heuristique h :

C'est une estimation de la valeur maximale que l'on pourrait encore ajouter au sac à dos en prenant les éléments restants qui ne sont pas placés encore, compte tenu des contraintes de capacité.

-g : représente la valeur courante qui est la somme des valeurs de chaque objet placés dans un des sacs.

Pseudo code :

Algorithme Aetoile :

Entrée : objet [] obj, Sacàdos [] sacs(état initiale).

Ouvert : pile de nœuds initialement vide.

Sortie : une solution si succès sinon échec.

debut

 startTime = currentTime()

 sol = Nouvelle liste de listes d'objets

 solcourante = 0

 sac = Nouvelle liste de listes d'objets

 // Initialisation des sacs

 POUR i DE 0 À taille(sacs) FAIRE

 sac[i] = Nouvelle liste d'objets

 // Initialisation du poids courant de chaque sac

 poidsCourant = Nouveau tableau de taille taille(sacs)

 POUR i DE 0 À taille(sacs) FAIRE

 poidsCourant[i] = 0

 // Calcul de la valeur f de l'élément

 f = CompterF(sacs, sac, obj, poidsCourant)

 // Création de l'élément initial pour la file de priorité

 element_pile = Nouvelle Pile(f, sac, poidsCourant)

 ouvert = Nouvelle file de priorité

 ouvert.ajouter(element_pile)

 trouve = faux

 TANT QUE (la file de priorité n'est pas vide ET trouvé est faux)

 // Retirer l'élément avec la plus petite valeur de f de la file de priorité

 element_pile = ouvert.retirer()

 // Calculer le nombre d'objets dans les sacs

 nb = 0

 POUR CHAQUE sac DANS element_pile.sacs FAIRE

 nb += taille(sac)

 // Générer les nouveaux états à partir de l'élément actuel

 SI nb ≠ taille(obj) ALORS

 nbfiles = 0

 nbfaux = 0

 POUR i DE 0 À taille(obj) FAIRE

 cpt = 0

 POUR CHAQUE sac DANS element_pile.sacs FAIRE

 SI sac NE contient PAS obj[i] ALORS

 cpt++

 FIN SI

 FIN POUR

 SI cpt == taille(element_pile.sacs) ALORS

 POUR CHAQUE sac DANS element_pile.sacs FAIRE

 nbfiles++

 sac1 = Copie(sac)

 sac1.ajouter(obj[i])

```

    SI (poidsCourant[sac] + obj[i].poids) > sacs[sac].poidsmax ALORS
        nbfaux++
    SINON
        poidcour = Copie(poidsCourant)
        poidcour[sac] += obj[i].poids
        sac2 = Copie(element_pile.sacs)
        sac2[sac] = sac1
        fnv = CompterF(sacs, sac2, obj, poidcour)
        element_pile2 = Nouvelle Pile(fnv, sac2, poidcour)
        ouvert.ajouter(element_pile2)
    FIN SI
FIN POUR
FIN SI
FIN POUR
SINON
    trouve = vrai
    solcourante = 0
    POUR CHAQUE sac DANS element_pile.sacs FAIRE
        POUR CHAQUE objet DANS sac FAIRE
            solcourante += objet.valeur
        FIN POUR
    FIN POUR
    sol = Copie(element_pile.sacs)
FIN SI
FIN TANT QUE
endTime = currentTime()
duration = endTime - startTime
tempsEcouleEnSecondes = duration / 1_000_000_000.0
AFFICHER "Temps écoulé en secondes : " + tempsEcouleEnSecondes
SI solcourante == 0 ALORS AFFICHER "Pas de solution"
AFFICHER "La valeur totale : " + solcourante + " Le sac : "
FIN.

```

Fonctionnement de l'algorithme :

Initialement les sacs sont vides donc la liste on l'initialise à vide qui va représenter les sacs ensuite on empile l'élément on initialise le poids courant de chaque sac à 0 et je compte la fonction f, j'empile cet élément dans ouvert et je fais une boucle qui va se répéter tant que ouvert n'est pas vide ou bien on a trouvé une solution valide.

On rentre dans la boucle, on dépile l'élément de ouvert (qui a la fonction f maximale) et je calcule la taille des sacs, si la somme de ces dernières égale à nombre d'objets donc c'est une solution finale je vais l'évaluer et calculer la valeur totale, sinon je vais générer les fils de cet élément après avoir vérifié l'existence des objets ou non dans ce dernier et puis j'empile dans ouvert que les fils valides qui respectent les contraintes de capacités. Si tous les fils sont pas valide et donc la solution finale sera celui qui était dépiler de la pile ouvert et donc j'évalue la solution et je calcule la valeur totale.

Exemple du Fonctionnement :

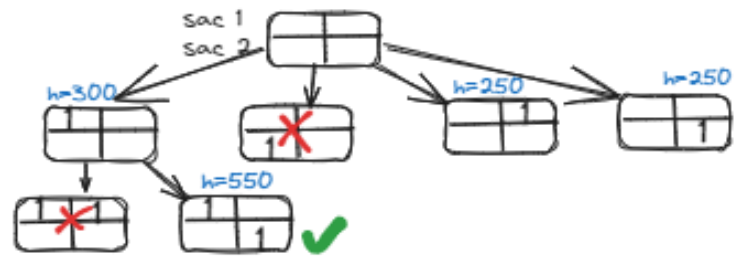
« Pour 2sacs et 2 objets »

Sac1 poidsmax : 250

Sac 2 poidsmax 120

Obj 1 $p=200, v=250$

Obj2 $p=88, v=300$



les elements representés par croix \times
ne sont pas dans ouvert(on va pas les traiter)

l'element qui a \checkmark veut dire que c'est la solution optimale

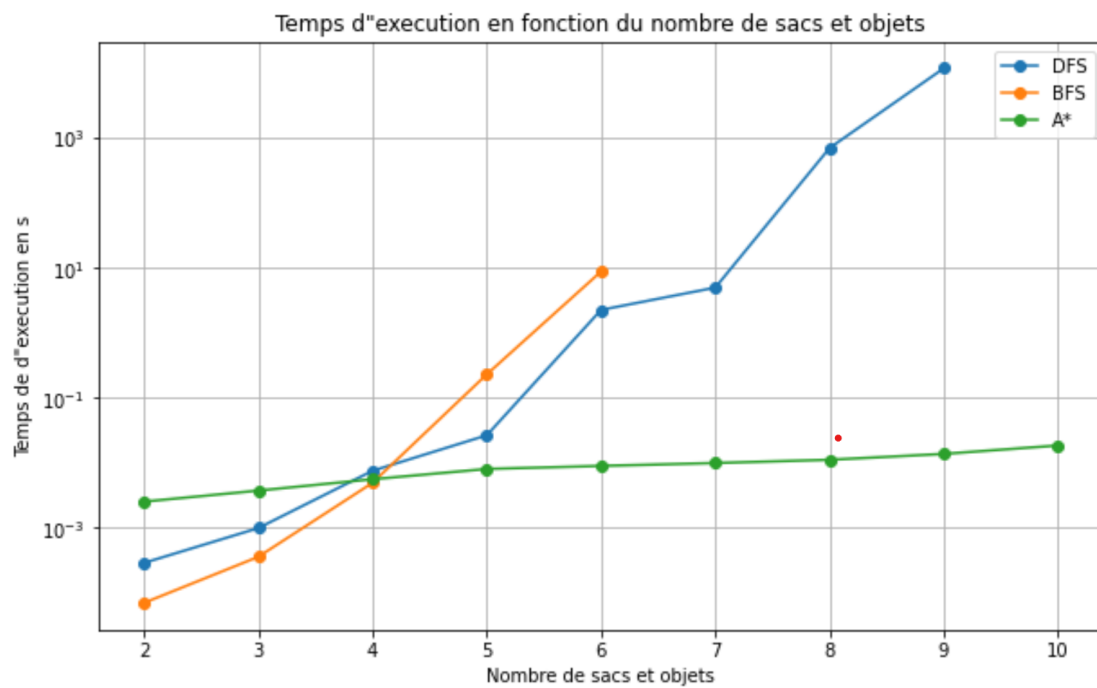
Expérimentation et comparaison :

A noter que les valeurs des objets et sacs sont les pour une instance donner pour les différent algorithme

-Tableaux démontrant le temps d'exécution des différent algorithme en seconde (nombre de sacs égal nombre d'objets) :

Nombre sacs et objets	2	3	4	5	6	7	8	9	10
DFS	2.894E-4	9.933E-4	0.0075606	0.0265866	2.2485922	4.9588092	11,5 min	3,25 h	/
BFS	7.03E-5	3.591E-4	0.0049331	0.2298464	8.7566852	/	/	/	/
A*	0.0045352	0.0037447	0.0056077	0.0080589	0.0089796	0.0099438	0.0111611	0.0137363	0.0185142

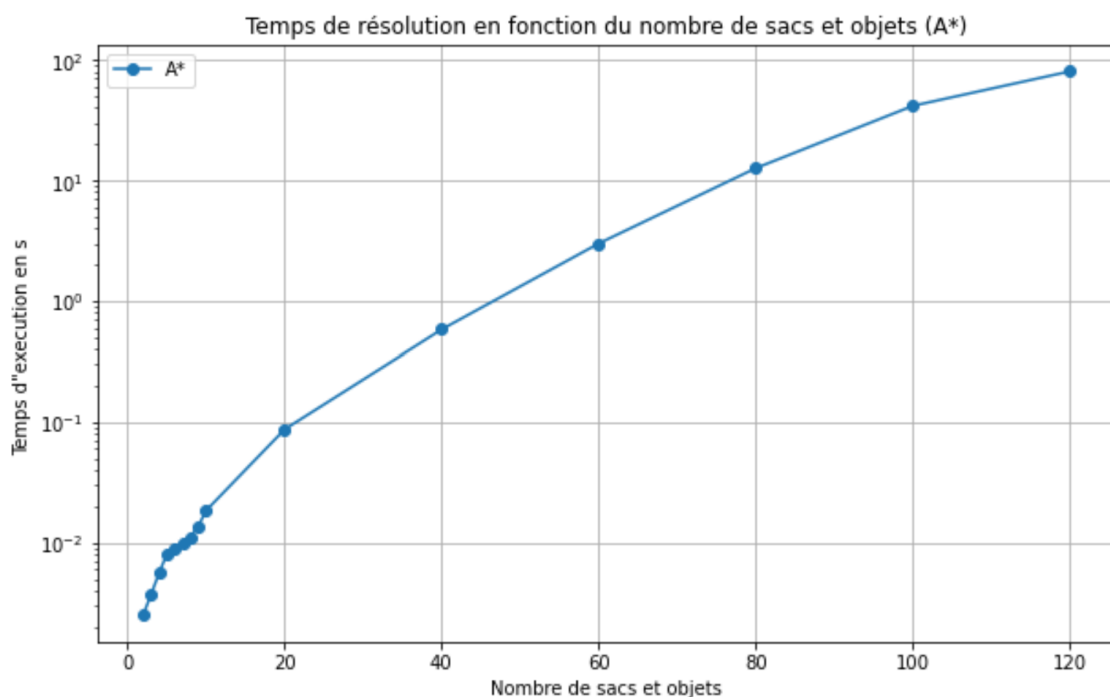
Graphe associe :



Pour algorithme A* plus de valeurs :

Nombre sacs et objets	20	40	60	80	100	120
A*	0.0872236	0.5816419	3.0006503	12.6472143	41.6090665	80.3742605

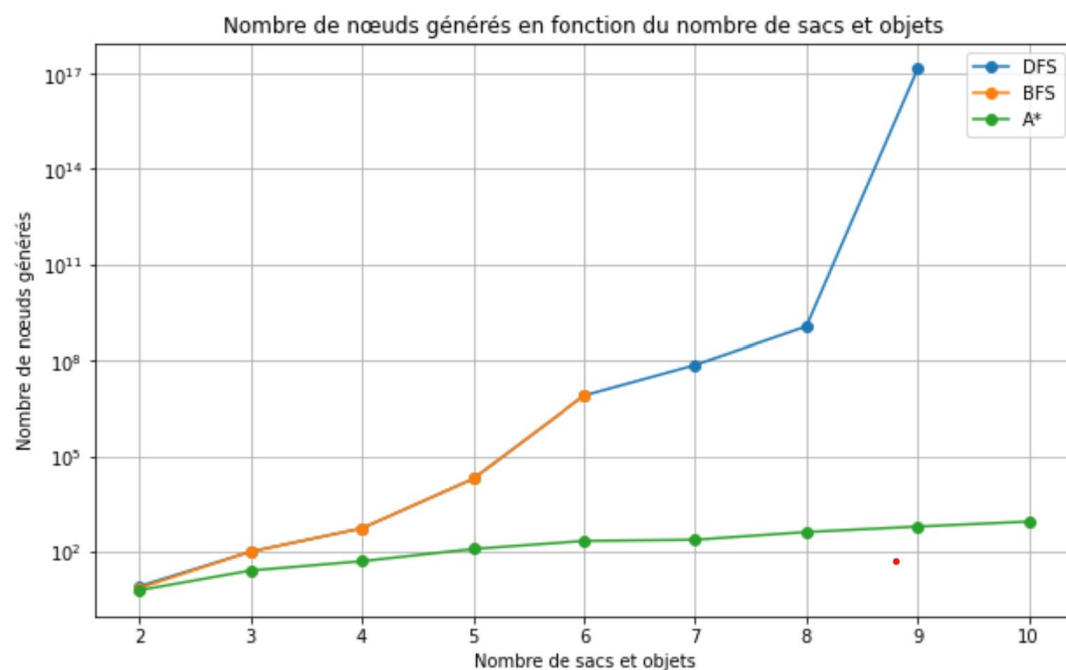
Graphe associe :



-Tableaux démontrant le **nombre de nœuds** développées des différent algorithme (nombre de sacs égal nombre d'objets) :

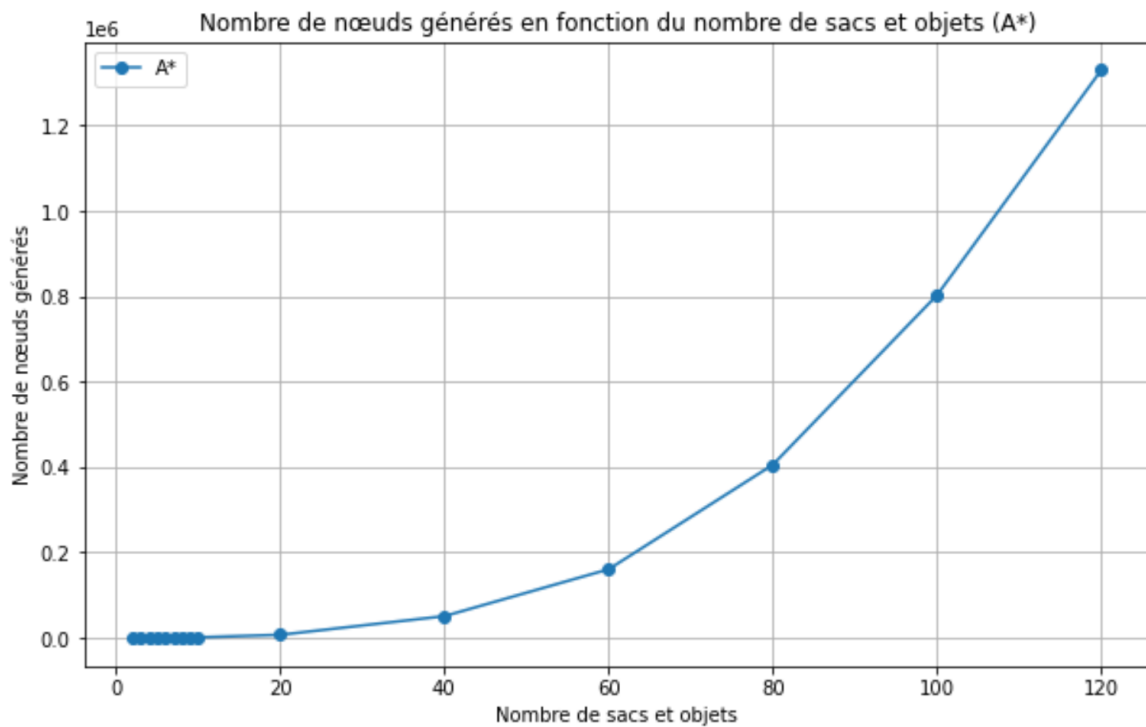
Nombre sacs t objets	2	3	4	5	6	7	8	9	10
DFS	8	98	532	19177	7930481	71736795	1138499190	140389813697084857	/
BFS	7	98	532	19177	7930481	/	/	/	/
A*	6	25	49	119	212	232	405	601	864

Graphe associé :



Pour algorithme A* plus de valeur :

Nombre sacs t objets	20	40	60	80	100	120
A*	6725	50500	160427	405403	802824	1329900



Tableaux démontrant le temps d'exécution des différents algorithmes en seconde (nombre de sacs varie et nombre d'objets varie) :

nb sacs	Nb objets	DFS	BFS	A*
6	4	0.1638002	0.2790817	0.0069502
7	4	0.0385464	0.0629392	0.0089349
8	4	0.0992063	0.1551179	0.0079197
9	4	0.1054176	0.1615813	0.0065378
10	4	0.4498275	2.1714353	0.0085519
11	4	0.7598097	6.923093	0.0093548
7	6	4.9951156	27.2575843	0.0084377
8	6	62.343547	/	0.0162357
5	3	0.021071	0.2906655	0.0626358
6	3	0.892792	0.892792	0.009226
7	3	0.2181676	0.8020628	0.01

Tableaux démontrant le nombre de nœud développe des différent algorithme (nombre de sacs varie et nombre d'objets varie) :

nb sacs	Nb objets	DFS	BFS	A*
6	4	87487	87487	105
7	4	5364	5364	62
8	4	5964	5964	71
9	4	279469	279469	112
10	4	767461	767461	160
11	4	42310	42310	85
7	6	9422965	9422965	240
8	6	6.1093795E7	/	300
5	3	3267	3267	64
6	3	49220	49220	95
7	3	98944	98944	112

Comparaison par rapport au temps d'exécutions et performance :

Les résultats obtenus montrent que pour des valeurs de sacs et d'objets inférieures à 4, BFS et DFS sont initialement plus rapides que A*. Cependant, au-delà de ces valeurs, A* devient plus performant et rapide. Il est intéressant de noter que, malgré cela, A* atteint le même résultat final que BFS et DFS.

De plus, BFS et DFS ne sont pas appropriés pour des valeurs de sacs et d'objets supérieures à 7 pour BFS et pour DFS on a pu aller jusqu'à 10, cependant, poursuivre au-delà de cette profondeur entraînerait des temps d'exécution plus longs, tandis qu'A* peut gérer des valeurs allant jusqu'à 120 sacs et objets dans un temps d'exécution acceptable.

En termes de performances, notamment en ce qui concerne le temps d'exécution et la gestion d'un grand nombre de valeurs, il est clair que A* est meilleur.

Comparaison par rapport au nombre de nœud développé :

Il est évident d'après les résultats que A* développe moins de nœuds que BFS et DFS, bien que ces derniers développent le même nombre de nœuds. Cette différence s'explique par le fait qu'A* est guidé par une heuristique, ce qui lui permet de choisir les nœuds les plus prometteurs en termes de coût total.

Comparaison par rapport au temps d'exécutions et performance (nombre de sacs varie et nombre d'objets varie) :

Les résultats obtenus montrent que A* plus rapides que BFS et DFS.

De plus, BFS et DFS ne sont pas appropriés pour des valeurs de sacs et d'objets grandes pour BFS et DFS, tandis qu'A* peut gérer des valeurs très grande sacs et objets.

Donc pour le temps d'exécution et la performance de nombre de valeur à gérer, clairement A* est meilleur.

Comparaison par rapport au nombre de nœud développé

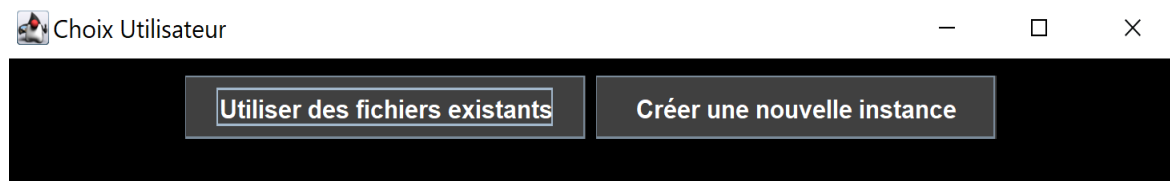
(Nombre de sacs varie et nombre d'objets varie):

Il est clair d'après les résultats que A* développe moins de nœuds que BFS et DFS, même si ces derniers développent le même nombre de nœuds. Cette différence s'explique par le fait qu'A* est guidé par une heuristique, ce qui lui permet de choisir les nœuds les plus prometteurs en termes de coût total.

Interface graphique

Scénario :

Demander à l'utilisateur d'utiliser des fichiers de sacs et d'objets déjà prêts ou de créer une nouvelle instance :



a) Le cas où le choix est de créer une nouvelle instance :

- 1- Demander les informations suivantes pour créer 2 nouveaux fichiers de sacs et objets.**

Poids maximum :	<input type="text"/>
Poids minimum :	<input type="text"/>
Nombre de sacs :	<input type="text"/>
Nombre d'objets :	<input type="text"/>
Valeur maximum :	<input type="text"/>
Valeur minimum :	<input type="text"/>


Après la création des 2 fichiers.

2- Demander quel algorithme souhaite-t-il utiliser :

Choisissez un algorithme

3- Afficher les résultats :

Choisissez un algorithme



La durée: 3224500ns La profondeur: 4
nombre de noeud: 532 La valeur: 35

Sac 1:
Poids max: 13, Poids courant: 0, Valeur courante: 0
Objets:

Sac 2:
Poids max: 12, Poids courant: 12, Valeur courante: 4
Objets:
Objet 1: Poids=12, Valeur=4

Sac 3:
Poids max: 16, Poids courant: 8, Valeur courante: 5
Objets:
Objet 3: Poids=8, Valeur=5

Sac 4:
Poids max: 19, Poids courant: 19, Valeur courante: 26
Objets:
Objet 2: Poids=6, Valeur=12
Objet 4: Poids=13, Valeur=14

b) Le cas où le choix est utilisé des fichiers existants :

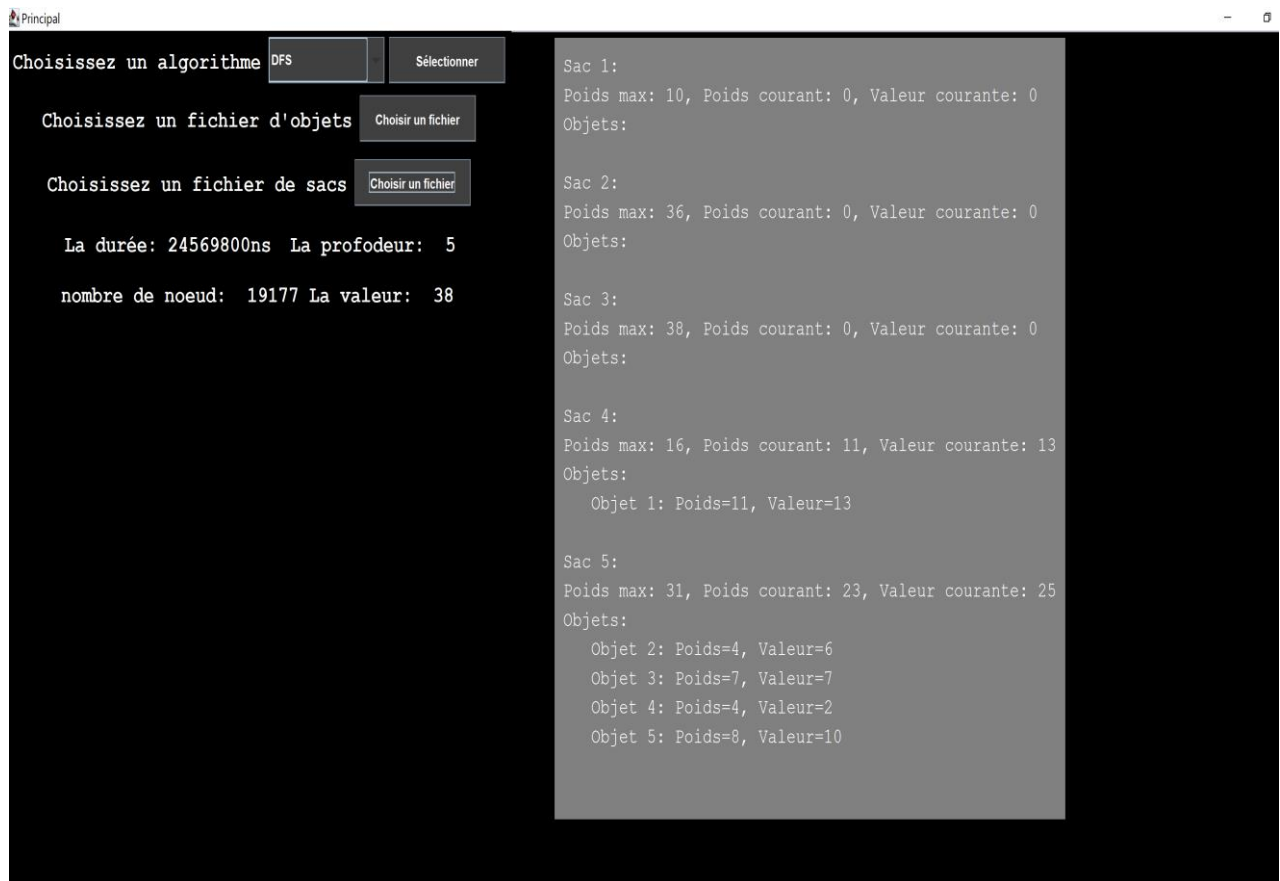
1- Demander de choisir l'algorithme, les fichiers de sacs et d'objets

Choisissez un algorithme

Choisissez un fichier d'objets

Choisissez un fichier de sacs

Puis afficher les résultats :



Conclusion :

En conclusion, le problème des sacs à dos multiples est un défi complexe et intéressant en informatique combinatoire. Bien qu'il soit difficile de résoudre exactement pour de grandes instances, il existe plusieurs approches algorithmiques telles que les méthodes précédemment examinées, il existe d'autres approches, notamment les métaheuristiques, qui peuvent être explorées pour obtenir des solutions de qualité dans un temps plus raisonnable.