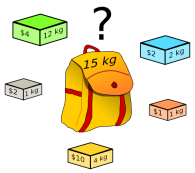
Projet d'algorithmique
Problème du sac à dos

En algorithmique, le problème du sac à dos est un problème d'optimisation combinatoire. Il modélise une situation analogue au remplissage d'un sac à dos, ne pouvant supporter plus d'un certain poids, avec tout ou partie d'un ensemble donné d'objets ayant chacun un poids et une valeur. Les objets mis dans le sac à dos doivent maximiser la valeur totale, sans dépasser le poids maximum.

Le problème du sac à dos est l'un des 21 problèmes NP-complets de Richard Karp, exposés dans son article de 1972. La formulation du problème est fort simple, mais sa résolution est plus complexe. Les algorithmes existants peuvent résoudre des instances pratiques de taille importante. Cependant, la structure singulière du problème, et le fait qu'il soit présent en tant que sous-problème d'autres problèmes plus généraux, en font un sujet de choix pour la recherche.



Quelles boîtes choisir afin de maximiser la somme emportée tout en ne dépassant pas les 15 kg autorisés ?

Présentation du problème

L'énoncé de ce problème fameux est simple :

> *"Étant donné plusieurs objets possédant chacun un poids et une valeur et étant donné un poids maximum pour le sac, quels objets faut-il mettre dans le sac de manière à maximiser la valeur totale sans dépasser le poids maximal autorisé pour le sac ?"*

Toute formulation de ce problème commence par un énoncé des données. Dans notre cas, nous avons un sac à dos de poids maximal P et n objets. Pour chaque objet i, nous avons un poids p_i et une valeur v_i.

Pour quatre objets ($n = 4$) et un sac à dos d'un poids maximal de 30 kg ($P = 30$), nous avons par exemple les données suivantes :

Objets	1	2	3	4
v_i	7	4	3	3
p_i	13	12	8	10

Ensuite, il nous faut définir les variables qui représentent en quelque sorte les actions ou les décisions qui amèneront à trouver une solution. On définit la variable x_i associée à un objet i de la façon suivante : $x_i = 1$ si l'objet i est mis dans le sac, et $x_i = 0$ si l'objet i n'est pas mis dans le sac.

Dans notre exemple, une solution réalisable est de mettre tous les objets dans le sac à dos sauf le premier, nous avons donc : $x_1 = 0, x_2 = 1, x_3 = 1$, et $x_4 = 1$.

Puis il faut définir les contraintes du problème. Ici, il n'y en a qu'une : la somme des poids de tous les objets dans le sac doit être inférieure ou égale au poids maximal du sac à dos. Cela s'écrit ici $x_1.p_1 + x_2.p_2 + x_3.p_3 + x_4.p_4 \leq P$ et pour n objets :

$$\sum_{i=1}^{n} x_i.p_i \leq P$$

Pour vérifier que la contrainte est respectée dans notre exemple, il suffit de calculer cette somme : $0 \times 13 + 1 \times 12 + 1 \times 8 + 1 \times 10 = 30$, ce qui est bien inférieur ou égal à 30, donc la contrainte est respectée. **Nous parlons alors de solution réalisable**. Mais ce n'est pas nécessairement la meilleure solution.

Enfin, il faut exprimer la fonction qui traduit notre objectif : maximiser la valeur totale des objets dans le sac. Pour n objets, cela s'écrit :

$$\max \sum_{i=1}^{n} x_i.v_i \leq P$$

Dans notre exemple, la valeur totale contenue dans le sac est égale à 10. Cette solution n'est pas la meilleure, car il existe une autre solution de valeur plus grande que 10 : il faut prendre seulement les objets 1 et 2 qui donneront une valeur totale de 11. Il n'existe pas de meilleure solution que cette dernière, nous dirons alors que **cette solution est optimale**.

Méthodes de résolution

Il existe deux grandes catégories de méthodes de résolution de problèmes d'optimisation combinatoire : les méthodes exactes et les méthodes approchées. Les méthodes exactes permettent d'obtenir la solution optimale à chaque fois, mais le temps de calcul peut être long si le problème est compliqué à résoudre. Les méthodes approchées, encore appelées heuristiques, permettent d'obtenir rapidement une solution approchée, donc pas nécessairement optimale.

Méthode approchée (version gloutonne)

Une méthode approchée a pour but de trouver une solution avec un bon compromis entre la qualité de la solution et le temps de calcul. Pour le problème du sac à dos, voici un exemple d'algorithme de ce type :

— calculer le rapport (v_i/p_i) pour chaque objet i ;
— trier tous les objets par ordre décroissant de cette valeur ;
— sélectionner les objets un à un dans l'ordre du tri et ajouter l'objet sélectionné dans le sac si le poids maximal reste respecté.

Déroulons cet algorithme sur notre exemple :

Première étape :

Objets	1	2	3	4
v_i/p_i	0.54	0.33	0.37	0.30

Deuxième étape :

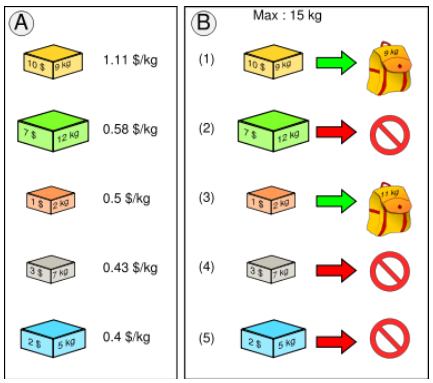
Objets	1	3	2	4
v_i	7	3	4	3
p_i	13	8	12	10
v_i/p_i	0.54	0.37	0.33	0.30

Troisième étape :

— Objet 1 : on le met dans le sac vide, le poids du sac est alors de 13 et inférieur à 30 ;
— Objet 3 : on le met dans le sac car $13 + 8 = 21$ est inférieur à 30 ;
— Objet 2 : on ne le met pas dans le sac car le poids total (33) dépasserait 30.
— Objet 4 : on ne le met pas dans le sac (poids total de 31).

La solution trouvée est donc de mettre les objets 1 et 3 dans le sac, ce qui donne une valeur de 10. Cette solution n'est pas optimale, puisqu'une solution avec une valeur totale de 11 existe. Néanmoins, cet algorithme reste rapide même si le nombre d'objets augmente considérablement.

Voici l'illustration d'un autre exemple :



Les deux phases de l'algorithme glouton. À gauche : tri des boîtes par ordre d'intérêt (ici en dollars par kilogramme). À droite : insertion dans l'ordre des boîtes, si cela est possible. On obtient ici une solution de 11 $ pour 11 kg alors que la solution optimale est de 12 $ et 14 kg.

Ce type d'algorithme est aussi appelé **algorithme glouton**, car il ne remet jamais en cause une décision prise auparavant. Ici, lorsque l'objet 2 ne peut pas entrer dans le sac, l'algorithme n'essaie pas d'enlever l'objet 3 du sac pour y mettre l'objet 2 à sa place.

Méthodes exactes

Pour trouver la solution optimale, et être certain qu'il n'y a pas mieux, il faut utiliser une méthode exacte, qui demande un temps de calcul beaucoup plus long (si le problème est difficile à résoudre). Il n'existe pas une méthode exacte universellement plus rapide que toutes les autres. Chaque problème possède des méthodes mieux adaptées que d'autres.

1) Programmation dynamique Le problème du sac à dos possède la propriété de sous-structure optimale, c'est-à-dire que l'on peut construire la solution optimale du problème à i variables à partir du problème à $i-1$ variables. Cette propriété permet d'utiliser une méthode de résolution par **programmation dynamique**.

La programmation dynamique s'appuie sur un principe simple, appelé le principe d'optimalité de Bellman : toute solution optimale s'appuie elle-même sur des sous-problèmes résolus localement de façon optimale. Concrètement, cela signifie que l'on peut déduire une ou la solution optimale d'un problème en combinant des solutions optimales d'une série de sous-problèmes. Les solutions des problèmes sont calculées de manière ascendante, c'est-à-dire qu'on débute par les solutions des sous-problèmes les plus petits pour ensuite déduire progressivement les solutions de l'ensemble.

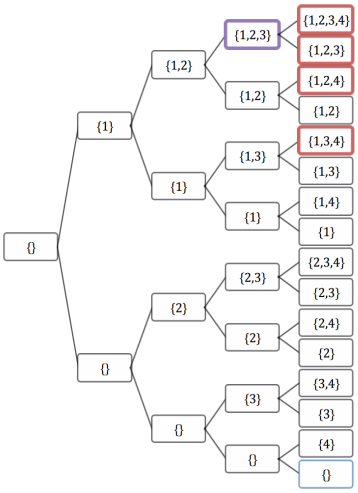
Ce paradigme algorithmique n'étant pas étudié dans le cadre du module AAV, une introduction à la programmation dynamique (en auto-formation) est disponible à l'adresse suivante :
`https://openclassrooms.com/courses/introduction-a-la-programmation-dynamique`
Ce cours en ligne explique également comment appliquer ce paradigme de programmation à la résolution du problème du sac à dos.

2) Procédure par séparation et évaluation (PSE) Nous allons présenter un exemple d'algorithme de recherche de solution optimale, nommé procédure par séparation et évaluation (PSE), ou en anglais *branch and bound*. Nous n'exposerons ici qu'une version simplifiée d'une PSE.

Une PSE est un algorithme qui permet d'énumérer intelligemment toutes les solutions possibles. En pratique, seules les solutions potentiellement de bonne qualité seront énumérées, les solutions ne pouvant pas conduire à améliorer la solution courante ne sont pas explorées.

Pour représenter une PSE, nous utilisons un "arbre de recherche" constitué :
— de nœuds ou sommets, où un nœud représente une étape de construction de la solution ;
— d'arcs pour indiquer certains choix faits pour construire la solution.

Dans notre exemple, les nœuds représentent une étape pour laquelle des objets auront été mis dans le sac, d'autres auront été laissés en dehors du sac, et d'autres, encore, pour lesquels aucune décision n'aura encore été prise. Chaque arc indique l'action de mettre un objet dans le sac ou, au contraire, de ne pas le mettre dans le sac. La figure suivante représente l'arbre de recherche du problème donné en exemple.



On associe l'ensemble vide à la racine (dont la profondeur est 0) et pour un nœud de profondeur $i-1$, on construit deux fils : celui du haut où l'on ajoute l'objet i dans le sac et celui du bas où le sac reste tel quel. L'arbre de recherche achevé, chaque feuille représente alors une solution potentielle mais pas forcément réalisable. Dans le schéma, les feuilles au bord épais représentent les propositions irréalisables car supérieures au poids maximal à ne pas dépasser. Pour déterminer la solution, il suffit de calculer la valeur du sac pour chaque nœud feuille acceptable et de prendre la solution ayant la plus grande valeur.

Cependant la taille de l'arbre de recherche est exponentielle en le nombre d'objets. Aussi il existe de nombreuses techniques algorithmiques de parcours de ce type d'arbre. Ces techniques ont pour but de diminuer la

taille de l'arbre et d'augmenter la rapidité du calcul. Par exemple, on peut remarquer que le poids du nœud interne $1, 2, 3$ dépasse déjà le poids maximal, il n'était donc pas nécessaire de développer l'étape suivante avec l'objet 4. Les procédures par séparation et évaluation (PSE) permettent d'élaguer encore plus cet arbre en utilisant des bornes inférieures et supérieures de la fonction objectif :

- Une **borne inférieure** est une valeur minimum de la fonction objectif. Autrement dit, c'est une valeur qui est nécessairement inférieure à la valeur de la meilleure solution possible. Dans notre cas, une configuration du sac réalisable quelconque fournit une borne inférieure.
- Une **borne supérieure** est une valeur maximale de la fonction objectif. Autrement dit, nous savons que la meilleure solution a nécessairement une valeur plus petite. Dans notre cas, nous savons que la valeur de la meilleure solution est nécessairement inférieure à la somme des valeurs de tous les objets (comme si on pouvait tous les mettre dans le sac).

Supposons maintenant que la borne inférieure soit initialisée par l'algorithme heuristique vu précédemment. Pendant la recherche à chaque nœud, nous pouvons déterminer une borne supérieure : la somme de toutes les valeurs de tous les objets déjà mis dans le sac plus la somme des valeurs des objets restants dont on ne sait pas encore s'ils seront dans le sac. À partir d'un nœud et de sa borne supérieure, nous savons que les solutions descendantes de ce nœud ne pourront pas avoir une valeur plus grande que cette borne supérieure. Si jamais, à un nœud donné, la valeur de la borne supérieure est inférieure à la valeur de la borne inférieure, alors il est inutile d'explorer les nœuds descendants de celui-ci. On dit qu'on **coupe l'arbre de recherche**. En effet, si à partir d'un nœud, nous savons que nous ne pourrons pas faire plus de 10 (borne supérieure calculée) et que la borne inférieure existante est à 11 (on a déjà une solution de valeur 11), alors les solutions descendantes de ce nœud ne sont pas intéressantes. Enfin, dernière remarque, la valeur de la borne inférieure peut être actualisée lorsqu'est trouvée une solution réalisable qui possède une valeur plus grande.

Ce système de calcul de bornes demande un faible coût de temps de calcul, et permet d'augmenter la rapidité de la PSE puisqu'elle coupe des branches d'arbre pour ne pas perdre de temps à les explorer. D'autres techniques servent à diminuer la taille de l'arbre et à augmenter la rapidité. Par exemple, elles sont basées sur l'ordre dans lequel on prend les décisions sur les objets, ou sur une évaluation à chaque nœud, ou encore sur des propriétés du problème qui permettent de déduire des conclusions sur certains objets. Sur notre exemple initial, il est possible d'obtenir de façon exacte la solution optimale en n'examinant que 9 nœuds au lieu de 31.

But du projet

Il s'agit d'implémenter puis de comparer les 3 méthodes décrites précédemment pour résoudre des instances du problème du sac à dos :

- (1) méthode approchée gloutonne ;
- méthodes exactes (2) par programmation dynamique et (3) par PSE.

Instructions

Vous utiliserez le langage Java pour implémenter les algorithmes utilisés dans le cadre de ce projet.

Le premier objectif est de se focaliser rapidement (dès la première semaine) sur une implémentation de la méthode gloutonne. Dans un premier temps vous pourrez vous concentrer sur la mise au point des algorithmes : l'énoncé des données pourra être contenu dans un tableau constant et le code source pourra être concentré dans un seul fichier. Ensuite, dans la version finale, vous proposerez une organisation modulaire des sources afin d'améliorer votre code et la qualité de votre travail en terme de lisibilité, maintenabilité, portabilité, extensibilité et ré-utilisabilité. Une fois la méthode gloutonne fonctionnelle, vous implémenterez les méthodes exactes en commençant par celle reposant sur la programmation dynamique.

Il s'agira aussi de proposer un programme de démonstration (type script) dont le synopsis est le suivant :

```
$>resoudre-sac-a-dos   chemin   poids-maximal   methode
```

où

- chemin : le chemin d'un fichier texte (voir exemple ci-dessous) contenant l'énoncé du sac à dos à résoudre, chaque ligne contiendra le nom d'un objet (une séquence d'au plus 30 caractères sans espace au sens large), son poids et sa valeur (deux nombres réels) ;
- poids_maximal : un nombre réel ;
- methode : choix de la méthode de résolution : gloutonne, prog. dynamique, pse.

Voici un exemple de fichier texte (`items.txt`) contenant l'énoncé du sac à dos à résoudre (1 ligne par objet) :

```
Lampe ; 2.0 ; 30.0
Sac de couchage ; 1.0 ; 20.0
Camping gaz ; 3.0 ; 40.0
Couteau suisse ; 0.2 ; 50.0
```

```
Snickers ; 0.1 ; 3.0
Tente 2 secondes ; 3.0 ; 100.0
Briquet ; 0.2 ; 3.0
Coca ; 10.0 ; 20.0
Chips ; 0.2 ; 2.0
```

Consignes Définir une classe `SacADos.java` permettant de résoudre le problème du sac a dos de différentes manières. Définir un constructeur `SacADos()` générant un sac vide. Définir un constructeur `SacADos(String chemin, float poids-maximal)` générant un sac vide, stockant la liste des objets possibles (contenus dans le fichier à l'adresse "chemin") et le poids maximal autorisé du sac. Surcharger la méthode `toString()` pour l'affichage du contenu du sac à dos. Définir des méthodes `resoudre()` qui permettront de résoudre le problème du remplissage du sac à dos de différentes manières.

Remarques

Projet à faire en binôme ou monôme **(pas de trinôme)**. **L'évaluation portera principalement sur la partie algorithmique du projet.** Évaluation et présentation des projets en dernière semaine de TD. A l'issue de la présentation, le code source (.zip) devra être envoyé par email à votre enseignant de TP. Pour la présentation, chaque binôme ou monôme devra apporter une version compilée du projet (**qui fonctionne sur les machines de l'IUT**) et réaliser une démonstration du programme devant un encadrant de TD.
⇒ Toute absence injustifiée entraînera une note nulle au projet.
⇒ **Afin de détecter une triche potentielle, les projets seront comparés via un outil automatique de détection de fraudes.**