TP - Projet de C - “Sac à Dos”

Yannick Kergosien

---

1 Préambule

L’objectif pédagogique de ce projet est multiple :
— renforcer votre connaissance et votre pratique en C,
— développer vos notions d’algorithmes,
— découvrir une partie de la Recherche Opérationnelle (même si les méthodes présentées dans le cadre des TPs ne sont pas adaptées à la résolution de ce problème).

Aucune correction ne sera donnée. L’évaluation portera sur la conformité, la qualité et la quantité du code produit pour répondre à la problématique posée ainsi que votre rapport et votre comportement pendant les TPs.

L’objectif du projet est de développer plusieurs méthodes de résolution pour un problème combinatoire connu, le problème du sac à dos. Ces méthodes de résolution seront à implémenter en C dans un seul programme. La suite du document décrit le projet à réaliser.

2 Le problème à résoudre

Le problème à résoudre est une généralisation du problème du sac à dos classique dont l’énoncé est simple : soit un sac à dos avec une capacité donnée (poids maximum), et un ensemble d’objets caractérisés par un poids et une valeur, l’objectif du problème est de trouver le sous ensemble d’objets à mettre dans le sac en maximisant la valeur total et sans dépasser la capacité.

Le problème à résoudre est nommé le problème du sac à dos multidimensionnel. L’énoncé est le même que précédemment sauf que le sac à dos possède plusieurs dimensions ($\approx$ capacités) tout comme les objets ont plusieurs dimensions ($\approx$ poids). La fonction objectif est toujours la même, maximiser la valeur total, et il faut respecter la contrainte de capacité de chacune des dimensions.

Les données du problème sont les suivantes :
— M : le nombre de dimension.
— N : le nombre d’objet avec pour chaque objet $j \in \{1, ..., N\}$:
  — p_j sa valeur,
  — $r_{i,j}$ sa dimension $i \in \{1, ..., M\}$.
— b_i : la dimension $i \in \{1, ..., M\}$ du sac à ne pas dépasser.

Mathématiquement, le problème s’écrit :

$$Max \sum_{j=1,...,N} p_j x_j$$

$$sous\ contrainte : \sum_{j=1,...,N} r_{i,j} x_j \leq b_i \quad \forall i = 1, ..., M$$

$$x_j \in 0, 1$$

3 Architecture du programme

Les éléments importants du programme à développer sont les suivants :
— un **parseur** qui a pour but de lire les données du problème (contenues dans un fichier texte) et de les stocker dans une structure **instance**.
— une structure **solution** qui permet de stocker une solution de ce problème.
— un panel de **méthodes de résolution** qui, à partir d'une instance, trouve la meilleure solution possible.
— une **sortie** qui a pour objectif d'écrire dans un fichier texte la meilleure solution trouvée pour une instance et d'écrire dans un autre fichier la valeur de la fonction objectif (valeur totale) et le temps total de résolution pour chaque instance. Ces deux derniers critères permettent d'évaluer l'efficacité d'une méthode.

Les paramètres lors de l'appel du programme sont les suivants :

1. le chemin vers le fichier de données correspondant à 30 instances à résoudre,
2. le numéro de codage à utiliser (point décrit plus loin),
3. le numéro de la méthode de résolution à utiliser (point décrit plus loin),
4. les paramètres de la méthode de résolution sélectionnée.

L'exécution du programme permet de résoudre les 30 instances une à une, d'écrire les 30 meilleures solutions trouvées dans des fichiers textes (une par fichier) et d'écrire la fonction objectif de chaque solution et le temps total de résolution dans un même fichier (1 ligne par instance).

4 Le Parseur

Le format d'un fichier de données est le suivant :
— le nombre d'instance (=30),
— pour chaque instance :
  — le nombre d'objet N, le nombre de dimension M, la valeur d'une solution trouvée (à ignorer), et une autre solution trouvée (à ignorer),
  — la valeur des variables x_j d'une solution réalisable (à ignorer),
  — la valeur p_j des objets,
  — pour chaque dimension, la valeur des poids $r_{i,j}$
  — les poids b_i à ne pas dépasser pour chaque dimension du sac.

Votre première tâche est de créer un projet sous Code Blocks ou Visual Studio. Puis vous devez commencer à implémenter le coeur de votre programme :
— lecture des 3 premiers paramètres,
— pour chaque instance :
  — lancer un timer,
  — lire le fichier et instancier la structure instance
  — afficher le nombre d'objet et le nombre de dimension de l'instance
  — arrêter le timer
  — écrire dans un fichier sortie.txt le temps écoulé entre le début et l'arrêt du timer
A noter que le fichier de sortie devra contenir trois colonnes : le nom de l'instance suivi de son numéro, la fonction objectif de la meilleure solution trouvée ('-' dans un premier temps) et le temps total de résolution. A vous de penser à allouer et à libérer la mémoire nécessaire au bon moment.

5 Le codage d'une solution

Le codage d'une solution indique comment une solution est représentée/modélisée. Le choix de codage d'une solution peut ne pas être simple selon le problème étudié ou la méthode de résolution utilisée. Ce codage doit répondre à trois grands critères :

— Efficacité : pour évaluer une solution, modifier une solution, etc.
— Complétude : toutes les solutions du problème doivent être représentées.
— Connexité : toute solution peut être atteignable à partir d'une autre solution en réalisant des "opérations simples'" (dépend de la méthode de résolution).

Il existe deux grandes familles de codage : direct et indirect. Pour ce projet, vous devrez utiliser un codage de chaque famille.

5.1 Codage direct

Le codage direct est le plus simple à mettre en oeuvre puisqu'il permet d'obtenir la solution "directement", c'est à dire sans calcul supplémentaire. Pour ce problème, vous utiliserez un tableau binaire de taille N. Chaque booléen du tableau indique si l'objet est dans le sac à dos ou non.

Petite remarque à la question "pourquoi ne pas énumérer toutes les solutions pour trouver la solution optimale ?" : de ce codage, il est possible de déduire simplement le nombre total de solution réalisable et non-réalisable : 2^N*. Supposons qu'un programme exécuté sur une machine performante puisse énumérer 50 solutions en une seule seconde, alors pour une instance avec seulement 30 objets, il faudrait attendre presque... un an... pour obtenir la solution optimale par cette méthode.*

5.2 Codage indirect

Cette méthode implique un calcul à partir du codage de la solution (décodeur) pour obtenir la solution réelle. Il existe plusieurs avantages d'utiliser ce type de codage : prise en compte de contraintes complexes, réduction de l'espace de recherche, simplifier certaines opérations des métaheuristiques, etc.

Pour ce problème, vous utiliserez une "permutation" de N éléments, c'est à dire un tableau de taille N contenant des entiers distincts (sans doublon) compris entre 1 et N. Ce codage symbolise un ordre de priorité d'objet à mettre dans le sac. Cette permutation ne permet pas d'obtenir directement la solution, l'agorithme ci-dessous sera à implémenter pour décoder la solution.

Algorithme 1: Décodeur

Input : Tab : une permutation des N objets
Output : La solution

```
1  i ← 0
2  Sac ← ∅
3  while i ≠ N do
4  |   j ← Tab[i]
5  |   if j peut loger dans le sac then
6  |   |   Sac ← Sac ∪ j
7  |   end
8  |   i ← i + 1
9  end
10 return Sac
```

Votre deuxième tâche est de compléter votre projet en ajoutant des structures incluant des variables et des méthodes pour ces deux codages. Ces structures doivent posséder :

- des variables permettant la codification de la solution,
- la méthode de décodage pour le deuxième codage uniquement,
- une méthode permettant d'évaluer une solution (fonction objectif),
- une méthode testant la faisabilité de la solution,
- une méthode d'affichage de la solution,
- une méthode d'écriture d'une solution dans un fichier.

Afin de tester votre code, vous devez pour chaque instance initialiser une solution aléatoirement (ou autrement), l'évaluer, tester sa faisabilité, l'afficher et l'écrire dans un fichier texte.