TP - Projet de C - “Sac à Dos”

Sujet de Yannick Kergosien, encadré par Simon Moulard
simon.moulard@univ-tours.fr

---

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.

A **chaque séance**, pensez à :
— tester systématiquement vos fonctions,
— commenter votre code,
— tester votre programme avec Valgrind.

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 la méthode de résolution à utiliser (point décrit plus loin),
3. 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 consiste à créer un projet dans un environnement Unix (Code Blocks, CodeLite, etc.). Puis vous devez commencer à implémenter le coeur de votre programme :
  — ouverture d'un fichier contenant les instances,
  — lecture des 3 premiers paramètres du fichier,
  — pour chaque instance :
    — lancer un timer,
    — lire le fichier et instancier la structure instance,
    — afficher le nombre d'objets et le nombre de dimensions 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 direct d'une solution

Le codage d'une solution indique comment une solution est représentée/modélisée. La façon la plus intuitive de représenter une solution est d'utiliser un tableau binaire de taille N. Chaque booléen du tableau indique si l'objet est dans le sac à dos ou non. Pour cette représentation, on parle de **codage direct**, puisqu'elle permet d'obtenir la solution "directement", c'est à dire sans calculs supplémentaires.

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 solutions réalisables et non-réalisables : 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.*

Votre deuxième tâche est de compléter votre projet en ajoutant une structure incluant des variables et des méthodes pour représenter une solution. Cette structure doit posséder :
  — des variables permettant la codification de la solution,
  — 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.

6 Heuristiques

Les heuristiques sont des méthodes de résolution, assez simples, permettant de trouver des solutions réalisables rapidement. Contrairement aux métaheuristiques, ce sont des méthodes dédiées à un problème. Ces heuristiques peuvent être utilisées pour trouver une ou plusieurs solutions initiales qui serviront aux métaheuristiques. Ces heuristiques peuvent être déterministes ou stochastiques si des composantes aléatoires interviennent.

Pour ce projet, vous allez devoir développer plusieurs méthodes basées sur une heuristique par construction par liste. Une heuristique par construction signifie que les décisions prises pour construire la solution ne seront jamais remises en cause plus tard dans l'algorithme. La liste a pour objectif de définir l'ordre dans lequel les décisions seront prises. Le schéma général de l'algorithme, pour un codage direct, est le suivant :

Algorithme 1 : Base d'une heuristique

Input : Données
Output : La solution (tableau binaire : SolutionDirect)

```
1  L ← Ordonner tous les objets selon une stratégie
2  SolutionDirect ← [0, 0, ..., 0]
3  while L ≠ ∅ do
4  |   j ← Premier élément de L
5  |   Retirer j de L
6  |   if j peut loger dans le sac then
7  |   |   SolutionDirect[j] ← 1
8  |   end
9  |   Mise à jour de L
10 end
11 return SolutionDirect
```

Quelques remarques :

— La stratégie d'ordonnancement des objets peut être :
  1 Aléatoire.
  2 Selon la valeur des objets (décroissant).
  3 Selon un ratio entre la valeur des objets et la somme des poids de toutes les dimensions (décroissant).
  4 Selon un ratio entre la valeur des objets et le poids de la dimension la plus critique (décroissant).
  5 etc.

6 La mise à jour dynamique de la liste peut être intéressante dans certains cas. Par exemple, au fur et à mesure que le sac se remplit, on réordonne la liste en fonction de la stratégie de trie 4 et de la nouvelle dimension la plus critique.

Votre troisième tâche est de développer 6 heuristiques qui utilisent :
— les 4 stratégies d'ordonnancement proposées,
— 1 une nouvelle stratégie d'ordonnancement (à vous de proposer),
— 1 mise à jour dynamique de la liste telle que décrit précédemment.

Ces heuristiques seront considérées comme des méthodes de résolution de type 1 (2ème paramètre du programme). Le 3ème paramètre définit le type de stratégie d'ordonnancement. N'oubliez pas de bien tester vos algorithmes et de comparer vos résultats.

7 Métaheuristiques

Cette section présente quelques métaheuristiques qui seront appliquées à ce problème de sac à dos multidimensionnel. Tout comme les heuristiques, elles peuvent être déterministes ou stochastiques. Il existe deux grandes catégories de métaheuristiques :

— "Parcours" : la méthode explore l'espace de solutions en utilisant qu'une seule solution à la fois.

— "Population" : la méthode explore l'espace de solutions en utilisant plusieurs solutions à la fois.

Les méthodes peuvent s'appuyer sur des phases d'intensification et/ou de diversification ainsi que l'utilisation de mémoire.

7.1 La recherche locale

Cette première métaheuristique est un classique en RO pour sa simplicité. Tout d'abord, une solution initiale est calculée. Cette solution est ensuite nommée solution courante. A partir de cette solution courante, plusieurs solutions sont construites. Cet ensemble de solutions est appelé voisinage de la solution courante. Ce voisinage est déterminé grâce à un opérateur de voisinage qui définit comment à partir de la solution courante, les solutions du voisinage sont générées. L'étape suivante consiste à sélectionner la meilleure solution parmi l'ensemble des solutions du voisinage. La solution sélectionnée devient la solution courante. Ce processus est ensuite répété tant que la solution courante est améliorée. L'algorithme 2 présente les grandes étapes de la recherche locale.

Adaptation au problème du sac à dos :

— Calcul d'une solution initiale : une heuristique développée précédement, à vous de choisir.

— Eval() : calcul la fonction objectif d'une solution

— L'élément le plus important est comment générer le voisinage. Plusieurs opérateurs de voisinages sont possibles :

  — Codage direct : un opérateur pertinent serait l'utilisation de deux sous-opérateurs. Le premier consisterait à ajouter un objet dans le sac (=transformer un 0 en 1), il faudrait alors tester tous les cas (N) et ne garder que les solutions réalisables. Ces solutions réalisables constitueront la première partie des solutions du voisinage. Le deuxième sous-opérateur consisterait à échanger un objet dans le sac avec un autre hors du sac (= mettre un 0 en 1 et mettre un 1 en 0). La deuxième partie des solutions voisines serait alors constituée des solutions réalisables résultantes de tous les échanges possibles.

Votre quatrième tâche est de développer une recherche locale pour le codage direct. N'oubliez pas de bien tester votre algorithme et de comparer vos résultats.

Algorithme 2 : Recherche Locale

Input : Données
Output : La meilleure solution trouvée

```
1  SolutionCourante ← Calcul d'une solution initiale
2  SolutionBest ← SolutionCourante
3  f_Best ← Eval(SolutionCourante)
4  continue ← VRAI
5  f_prec ← f_courant
6  while continue do
7      f_BestVoisin ← 0
8      forall les mouvements possibles à partir d'un opérateur de voisinage do
9          SolutionVoisine ← Calcul de la solution voisine à partir du mouvement et de
             SolutionCourante
10         if Eval(SolutionVoisine)> f_BestVoisin then
11             SolutionBestVoisine ← SolutionVoisine
12             f_BestVoisin ← Eval(SolutionVoisine)
13         end
14     end
15     f_courant ← f_BestVoisin
16     SolutionCourante ← SolutionBestVoisine
17     if f_courant>f_Best then
18         f_Best ← f_courant
19         SolutionBest ← SolutionCourante
20     end
21     else
22         if f_courant<f_prec then
23             continue ← FAUX
24         end
25     end
26     f_prec ← f_courant
27 end
28 return SolutionBest
```

7.2 La recherche tabou

La recherche locale a pour inconvénient de s'arrêter assez rapidement dans un optimum local sans explorer une grande partie de l'espace des solutions. Pour pallier à ce problème, plusieurs extensions de cette méthode ont été proposées. Une des plus connues se nomme la recherche tabou. La structure de base de cet algorithme est la même que la recherche locale, cependant, quelques étapes viennent s'ajouter ou sont modifiées :

— Le critère d'arrêt de l'algorithme n'est plus "dès que la solution courante ne peut plus être améliorée" mais c'est un nombre maximum d'itérations de l'algorithme ou un nombre maximum d'itérations sans amélioration de la meilleure solution connue. Vous utiliserez ce dernier critère que l'on considérera comme premier paramètre de la méthode.

— Une mémoire est utilisée pour éviter de ré-explorer les mêmes solutions. Cette mémoire est appelée liste taboue. L'idée de cette liste est qu'elle contient toutes les solutions explorées et lorsque de nouvelles solutions sont calculées (calcul du voisinage), seules les solutions n'appartenant pas à la liste taboue sont considérées dans le voisinage. Cependant stocker toutes les solutions explorées coûterait beaucoup trop de temps à l'algorithme. Il est communément accepté de stocker uniquement l'opération qui a permis d'obtenir la meilleure solution du voisinage. Ainsi la liste taboue va contenir des mouvements et ces mouvements seront interdis lors de la prochaine exploration du voisinage. Enfin pour éviter d'interdire l'ensemble des mouvements tout le long de l'algorithme (et s'interdire l'exploration de nouvelles solutions non explorées), la liste taboue a une taille fixe qui sera le deuxième paramètre de la méthode. Un nouveau mouvement déclaré comme tabou remplacera le plus ancien mouvement stocké dans la liste. Ainsi un mouvement sera considéré tabou un certain nombre d'itération de l'algorithme égale à la taille de la liste taboue.

— Enfin, un critère d'aspiration peut également être inclus. Si ce critère est activé alors lors de l'exploration du voisinage, si une solution voisine améliore la meilleure solution connue mais est calculée à partir d'un mouvement tabou alors elle est tout de même considérée. Si ce critère n'était pas activé alors cette solution ne serait pas retenue dans l'ensemble des solutions voisines. L'activation ou non du critère d'aspiration est le troisième et dernier paramètre de la méthode.

L'algorithme 3 présente les grandes étapes de la recherche tabou.

De nombreuses étapes peuvent être reprises de la recherche locale (calcul d'une solution initiale, les opérateurs de voisinages, etc.). Le dernier élément à définir est les mouvements que va contenir la liste taboue :

— Codage direct : un mouvement est défini par soit le numéro d'un objet dont le bit correspondant est passé de 0 à 1 soit la paire de numéros d'objets échangés.

Votre cinquième tâche est de développer une recherche tabou. N'oubliez pas de bien tester vos algorithmes, de trouver des valeurs pertinentes pour les paramètres et de comparer vos résultats.

Algorithme 3 : Recherche Tabou

Input : Données + $NbIteMax$ (nombre maximum d'itérations sans amélioration de la meilleure solution connue) + $TailleListe$ (la taille de la liste taboue) + $Aspi$ (Vrai si le critère d'aspiration est activé, faux sinon)

Output : La meilleure solution trouvée

```
1  SolutionCourante ← Calcul d'une solution initiale
2  SolutionBest ← SolutionCourante
3  f_Best ← Eval(SolutionCourante)
4  i ← 0
5  while i < NbIteMax do
6      f_BestVoisin ← 0
7      forall les mouvements possibles à partir d'un opérateur de voisinage do
8          if (le mouvement n'est pas tabou) ou (Aspi) then
9              SolutionVoisine ← Calcul de la solution voisine à partir du mouvement et de
                SolutionCourante
10             if le mouvement n'est pas tabou then
11                 if Eval(SolutionVoisine)> f_BestVoisin then
12                     SolutionBestVoisine ← SolutionVoisine
13                     f_BestVoisin ← Eval(SolutionVoisine)
14                     MouvementUtil ← Le mouvement utilisé
15                 end
16             end
17             else
18                 if Eval(SolutionVoisine)> f_Best then
19                     SolutionBestVoisine ← SolutionVoisine
20                     f_BestVoisin ← Eval(SolutionVoisine)
21                     MouvementUtil ← Le mouvement utilisé
22                 end
23             end
24         end
25     end
26     f_courant ← f_BestVoisin
27     SolutionCourante ← SolutionBestVoisine
28     Ajouter MouvementUtil à la liste taboue sans dépasser TailleListe
29     if f_courant>f_Best then
30         f_Best ← f_courant
31         SolutionBest ← SolutionCourante
32         i ← 0
33     end
34     i ← i + 1
35 end
36 return SolutionBest
```

7.3 L'algorithme génétique

Les algorithmes génétiques sont inspirés de la théorie de Darwin : "les individus les plus adaptés tendent à survivre plus longtemps et à se reproduire plus aisément". Le principe de cette méthode est d'explorer plusieurs solutions à la fois (chaque solution est aussi appelée individu). Tout d'abord, une population initiale d'individus (= un ensemble de solutions) est générée. Le nombre de solutions dans la population est un premier paramètre de la méthode noté $TaillePopu$. Puis cette population va évoluer pendant un certain nombre d'itération $NbIteMax$ (deuxième paramètre de la méthode). Cette évolution se déroule en trois grandes étapes :

- Sélection et croisement : des paires d'individus parents (solutions) vont être sélectionnées pour générer des paires d'individus enfants.
- Mutation : chaque enfant généré peut subir une mutation (un changement aléatoire) avec une certaine probabilité P_{mut} (troisième paramètre de la méthode).
- Renouvellement de la population : une nouvelle population est créée en fonction de l'ancienne population et de la nouvelle population enfant.

L'algorithme 4 présente les grandes étapes d'un algorithme génétique.

Adaptation de cet algorithme génétique au problème du sac à dos :

- Génération de la population initiale : l'idée de cette étape est de générer $TaillePopu$ solutions différentes en utilisant des heuristiques aléatoires (celles développées auparavant).
- Sélection des parents : l'idée est de sélectionner aléatoirement deux solutions parents de bonne qualité. Voici deux exemples :
  - Sélection par tournoi : 4 individus différents, notés A B C et D, sont aléatoirement sélectionnés de la population. La meilleure solution entre A et B devient le parent P1 et la meilleure solution entre C et D devient le parent P2.
  - Sélection par roulette : un individu est sélectionné avec une probabilité proportionnelle à la qualité de tous les individus de la population (ex : $P_{select}(s) = \frac{Eval(a)}{\sum_{i \in PopulationCourante} Eval(i)}$).
- Croisement d'enfants : cette étape consiste à créer deux enfants possédants des parties de solutions de chaque parent. Plusieurs possibilités selon le codage :
  - Codage direct : 1-point ou 2-points + une procédure de réparation (cf. exemple au tableau).
- Mutation : l'idée de cette étape est d'apporter de la diversification dans la recherche. Si un individu doit être muté alors un changement est réalisé selon le codage :
  - Codage direct : un objet est choisi aléatoirement puis le bit correspondant est changé (0 en 1 ou 1 en 0).
- Renouvellement de la population : l'idée est de garder les bonnes solutions sans pour autant garder uniquement les meilleures au risque de converger trop rapidement. Voici quelques possibilités :
  - Plus de parent : $PopulationCourante \leftarrow PopulationEnfant$
  - Elitiste : $PopulationCourante \leftarrow$ les $TaillePopu$ meilleures solutions de $PopulationCourante$ et $PopulationEnfant$
  - Equilibré : $PopulationCourante \leftarrow$ les $\frac{TaillePopu}{2}$ meilleures solutions de $PopulationCourante$ plus les $\frac{TaillePopu}{2}$ meilleures solutions de $PopulationEnfant$

Votre sixième tâche est de développer un algorithme génétique. A vous de choisir les étapes de sélection, croisement, mutation et renouvellement de la population. Vous pouvez également en implémenter plusieurs et tester différentes configurations grâce à des paramètres supplémentaires. N'oubliez pas de bien tester vos algorithmes, de trouver des valeurs pertinentes pour les paramètres et de comparer vos résultats.

Algorithme 4 : Algorithme génétique

Input : Données + $NbIteMax$ (nombre maximum d'itérations) + $TaillePopu$ (taille de la population) + P_{mut} (probabilité de mutation)
Output : La meilleure solution trouvée

$PopulationCourante \leftarrow$ Calcul d'une population initiale
$SolutionBest \leftarrow$ Meilleur individu de $PopulationCourante$
$f_{Best} \leftarrow$ Eval($SolutionBest$)
$i \leftarrow 0$
while $i < NbIteMax$ **do**
  $PopulationEnfant \leftarrow \emptyset$
  for $\frac{TaillePopu}{2}$ *itérations* **do**
    Sélectionner deux parents P1 et P2 dans $PopulationCourante$
    Croiser P1 et P2 afin d'obtenir deux enfants E1 et E2
    $PopulationEnfant \leftarrow PopulationEnfant \cup E1 \cup E2$
  end
  forall $Solution \in PopulationEnfant$ **do**
    if *Eval(*$Solution$*)>*f_{Best} **then**
      $f_{Best} \leftarrow$Eval($Solution$)
      $SolutionBest \leftarrow Solution$
    end
    if *l'individu doit être muté selon une probabilité* P_{mut} **then**
      $Solution \leftarrow$ Mutation($Solution$)
      Mise à jour de $PopulationEnfant$ en fonction de $Solution$
      if *Eval(*$Solution$*)>*f_{Best} **then**
        $f_{Best} \leftarrow$Eval($Solution$)
        $SolutionBest \leftarrow Solution$
      end
    end
  end
  $PopulationCourante \leftarrow$ Renouvellement($PopulationCourante$,$PopulationEnfant$)
  $i \leftarrow i + 1$
end
return SolutionBest

7.4 Une dernière : à vous de choisir

Pour la dernière méthode à implémenter, vous avez le choix entre :
— Implémenter une nouvelle méthode provenant :
  — de votre propre recherche (livre, web, etc.),
  — une idée demandée pendant le TP.
  Quelques mots clès : Recherche à voisinage variable (variable neighborhood search) ; recherche locale itérée (Iterated local search) ; recherche adaptative à voisinage large (An Adaptive Large Neighborhood Search Algorithm) ;...
— une nouvelle méthode que vous avez inventée (et justifiée).

Votre septième et dernière tâche est de développer une méthode de résolution de votre choix. Vous pouvez également en implémenter plusieurs et tester différentes configurations grâce à des paramètres supplémentaires. N'oubliez pas de bien tester vos algorithmes, de trouver des valeurs pertinentes pour les paramètres et de comparer vos résultats.

8 Rendu et évaluation

A la fin de votre projet, vous devez rendre vos sources et un rapport. Les critères d'évaluation ont déjà été présentés en préambule.

Vos sources doivent contenir uniquement un code qui compile sans erreur, commenté, propre et dont l'exécutable fonctionne. Le rapport présentera les fonctionnalités présentes et non-présentes. L'ensemble des sources doit être contenu dans un dossier zip nommé par vos noms (attention, pas d'exécutable dans le zip).

Le rapport doit contenir les éléments suivants :
- les outils utilisés (SE, compilateur et IDE).
- Une description de la structure / décomposition de votre programme (fichiers, structure de données, et fonctions).
- Pour chaque partie à développer, un compte rendu de ce qui a été réalisé, ou non, et une indication de l'état du code :
  - Codé à X%,
  - Testé à X%,
  - Fonctionne ou non,
  - Passe Valgrind ou non.
- Il est inutile de re-décrire les algorithmes de l'énoncé du TP, seuls les algorithmes (ou parties d'algorithmes) que vous proposez sont à décrire.
- Les tests que vous avez réalisés sur vos méthodes avec **une interprétation/analyse des résultats**.

Ne survendez surtout pas votre programme...

8.1 Principaux points évalués

Qualité du code :
- Décomposition du programme
- Décomposition en fonctions
- Propreté (indentation, nom des variables, ...)
- Fuites mémoires
- Commentaires

Fonctionnalités réalisées :
- Lecture/Écriture/interface utilisateur
- Codage direct
- Heuristiques
- Recherche locale
- Recherche tabou
- Algorithme génétique
- Méthode inventée

Qualité du rapport :
- Description de tous les éléments demandés
- Compréhension globale du sujet, prise de recul
- Analyse des résultats, justification des choix des paramètres
- Qualité de la rédaction
- Pénalités (e.g. retard)