



ÉCOLE POLYTECHNIQUE DE L'UNIVERSITÉ DE TOURS
64, Avenue Jean Portalis
37200 TOURS, FRANCE
Tél. (33)2-47-36-14-14
Fax (33)2-47-36-14-22
www.polytech.univ-tours.fr



DI3

Rapport de projet S4

Projet tutoré 2 : Sac à dos

Auteur(s)

Thomas Couchoud

[\[thomas.couchoud@etu.univ-tours.fr\]](mailto:thomas.couchoud@etu.univ-tours.fr)

Victor Coleau

[\[victor.coleau@etu.univ-tours.fr\]](mailto:victor.coleau@etu.univ-tours.fr)

Encadrant(s)

Yannick Kergosien

[\[yannick.kergosien@univ-tours.fr\]](mailto:yannick.kergosien@univ-tours.fr)

Polytech Tours

Département DI

Table des matières

1 Informations générales sur le projet	2
1.1 Outils utilisés	2
1.2 Structure du projet	2
2 Fichiers	4
2.1 Parser	4
2.2 Instance	4
2.3 Item	5
2.4 Bag	5
2.5 SolutionDirect	6
2.6 SolutionIndirect	6
2.7 Solution	7
2.8 Heuristic	7
2.9 Scheduler	8
2.10 MetaheuristicLocal	8
2.11 MetaheuristicTabou	9
2.12 MetaheuristicGenetic	9
2.13 MetaheuristicKaguya	9
Conclusion	10
Annexes	10

Introduction

Le projet que nous avons choisi est celui portant sur le problème du sac à dos (*Knapsack problem*). Celui-ci se généralise très simplement et donne lieu à de nombreux problèmes analogues.

Dans notre cas, nous devons remplir un sac à dos d'objets. Chaque objet a une certaine valeur prédéfinie ainsi que des "poids" dans différentes dimensions. On peut imaginer le cas où l'on tenterait de remplir sa valise pour partir en voyage. Chaque objet a une valeur selon l'importance qu'on lui donne ainsi que des "poids" qui pourraient être la place qu'il occupe, son poids réel etc.. L'idée ici est d'essayer de maximiser la valeur que nous emportons avec nous sachant que notre valise est limitée en poids et taille.

D'un point de vu mathématique, on peut modéliser ceci simplement :

- X Un vecteur définissant quels items sont dans le sac ou pas (ex : $(0; 0; 1)$ définira un sac avec seulement le 3ème item de pris).
- W_j Le poids maximum que le sac peut supporter dans la dimension i .
- $w_{i,j}$ Le poids du i ème item dans la j ème dimension.
- v_i La valeur du i ème item.

Les contraintes sont : $\forall j \in [0...m], \sum_{i=0}^n x_i w_{i,j} \leq W_j$

On appellera la fonction objectif $z(X)$ la fonction donnant la valeur d'un sac : $z(X) = \sum_{i=0}^m x_i v_i$

Chapitre 1

Informations générales sur le projet

1.1 Outils utilisés

Afin de réaliser notre projet, nous avons utilisé différents outils. Concernant les Systèmes d'exploitations, nous avons utilisé Windows (Victor), OSX (Thomas) et Ubuntu (Travis CI). Les IDEs sont : CodeBlocks, CLion, Atom, Notepad++. Le compilateur utilisé est gcc, un Makefile est disponible pour la compilation.

1.2 Structure du projet

Le C ayant un langage rassemblant tous les fichiers en un lors de la compilation, il est nécessaire de choisir judicieusement ses noms de fonctions afin d'éviter les duplicatas. Dans notre cas nous avons choisi un formatage simple : [Nom du .c / Nom de la structure]_[Nom de la fonction]. Nous aurons donc des fonctions du type *population_create(...)* ou bien *metaheuristicGenetic_search(...)*.

Concernant l'organisation des fichiers en eux même, chaque type d'entre eux est localisé à un endroit différent. En effet nous avons commencé par mettre tous nos .c et .h dans un même dossier. Cependant, le projet grandissant assez vite, il a rapidement arrivé un stade où l'on se perd. Pour cela nous avons décidé de séparer les .h des .c puisque nous travaillons principalement sur les .c. Cela permet de s'y retrouver plus aisément. Ainsi la structure de notre dossier source est la suivante :

- src → Le dossier racine contenant nos .c pour le programme
- headers → Le dossier contenant nos headers pour le programme
- unit → Le dossier contenant nos .c pour les tests unitaires
- headers → Le dossier contenant nos headers pour les tests unitaires

Intéressons-nous au dossier src, son contenu est :

- Parser → Regroupant les différentes fonctions afin de lire un fichier.
- Instance → Représentant une instance.
- Item → Représentant un élément de l'instance.
- Bag → Représentant le contenu du sac pour une solution indirecte.
- SolutionDirect → Représentant une solution directe.
- SolutionIndirect → Représentant une solution indirecte.
- Solution → Représentant l'union d'une solution directe et indirecte.
- Heuristic → Regroupant les fonctions liées à la résolution grâce à une heuristique.
- Scheduler → Regroupant les fonctions liées aux différents algorithmes pour l'heuristique.
- MetaheuristiqueLocal → Regroupant les fonctions liées aux différents algorithmes pour la metaheuristique locale.

- MetaheuristiqueTabou → Regroupant les fonctions liées aux différents algorithmes pour la metaheuristique tabou et contenant la structure Tabou.
- MetaheuristiqueGenetic → Regroupant les fonctions liées aux différents algorithmes pour la metaheuristique génétique et contenant la structure Population.
- MetaheuristicKaguya → Regroupant les fonctions liées à notre metaheuristique personnalisée (contenant les structures Clan, ClanMember et DNA).

Chapitre 2

Fichiers

Nous allons maintenant décrire brièvement les différentes fonctions contenus dans chaque fichier. Plus d'informations sont disponibles dans la doc des headers, notamment concernant les retours spéciaux (comme '-1' si un index n'est pas trouvé ou autres).

2.1 Parser - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

L'une des premières parties que nous devons réaliser est le parser. Lors de cette dernière, un choix important a du se faire : lisons-nous toutes les instances d'un fichier d'un seul coup ou lisons les nous une par une ?

Nous avons initialement décidé de les lire toutes à la suite. En effet, ce choix était celui de la simplicité. Nous avons voulu commencer simple afin de pouvoir avancer sans attendre sur les autres tâches à faire. Puis rapidement nous avons implémenté la seconde méthode. Celle-ci nous parut plus adéquate pour nos utilisations car elle permet d'éviter une utilisation importante de la mémoire pour pas grand chose. Certes nous avons du créer une structure Parser qui sert principalement à conserver les informations de la dernière lecture, mais ce choix nous parut être le meilleur.

Afin de satisfaire la seconde méthode, une structure Parser a été créée ainsi que ces fonctions associées :

- `parser_create` → Permet de créer cette structure à partir du chemin d'un fichier.
- `parser_destroy` → Pour détruire la structure.
- `parser_getNextInstance` → Renvoie la prochaine instance du fichier ou NULL si l'on a atteint la fin.

A côté de cela, des fonctions génériques sont présentes :

- `parser_readAllFile` → Renvoie toutes les instances d'un fichier.
- `parser_readInstance` → Renvoie une instance à partir d'un fichier ouvert à la bonne position.
- `parser_readLine` → Lit la prochaine ligne non vide, ou renvoie NULL si on a atteint la fin du fichier.
- `parser_lineToIntArray` → Convertit un string composé de nombres séparés par tabulation en un tableau d'entiers.
- `getline` → Lit la prochaine ligne du fichier.

2.2 Instance - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Le fichier Instance comporte une structure nommée Instance qui contient ces propriétés suivantes :

- itemsCount → Représentant le nombre d'items dans l'instance.
- dimensionsNumber → Représentant le nombre de dimensions dans l'instance.
- items → Un tableau d'Item (section 2.3) étant les éléments de l'instance.
- maxWeights → Un tableau d'entier représentant le poids maximum pour chaque dimension.

Les fonctions suivantes agissent toutes à partir d'une instance :

- instance_initialize → Permettant de créer une instance sur le tas. La fonction instance_setMaxWeights devra par la suite être appelé. Le tableau d'item est créé mais chaque item devra être initialisé grâce à item_setWeight.
- instance_getItem → Permet de récupérer un item à un index précis dans l'instance.
- instance_setMaxWeights → Permet de définir le tableau des poids maximums de l'instance. Le tableau doit être alloué sur le tas.
- instance_getMaxWeight → Permet de récupérer le poids maximum sur une dimension précise.
- instance_destroy → Détruit une instance précédemment créée par instance_initialize.
- instance_item_getWeight → Récupère le poids de l'item à un certain index dans l'instance.
- instance_item_getValue → Récupère la valeur de l'item à un certain index dans l'instance.

2.3 Item - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Le fichier Item contient une structure Item ayant pour propriétés :

- value → La valeur d'un item.
- weights → Un tableau de ses différents poids sur chaque dimension.

Les fonctions suivantes s'appliquent à partir d'une structure Item :

- item_initialize → Afin de créer un Item sur le tas.
- item_setWeight → Pour définir le poids d'un item dans la dimension souhaitée.
- item_getWeight → Pour obtenir le poids d'un item dans la dimension souhaitée.
- item_destroy → Afin de détruire un Item précédemment créé par item_initialize.

2.4 Bag - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Le fichier Bag contient une structure Bag permettant de stocker les indices des items pris dans notre sac. Ses propriétés sont :

- bag_create → Permet de créer un bag sur le tas à partir d'une instance.
- bag_destroy → Permet de détruire un bag précédemment créé par bag_create.
- bag_appendItem → Ajoute un item dans le sac.
- bag_canContain → Permet de savoir si un item va pouvoir rentrer dans le sac.
- bag_getItemIndex → Permet de récupérer l'indice de l'item à un index donné dans le sac.
- bag_getWeight → Récupère le poids actuel du sac dans la dimension demandée.

- `bag_addWeight` \rightarrow Ajoute du poids dans le sac dans la dimension donnée.
- `bag_saveItems` \leftarrow Ecrit le bag dans un fichier.
- `bag_print` \rightarrow Affiche le bag dans la console.
- `bag_getCriticDimension` \rightarrow Renvoie l'index de la dimension critique.
- `bag_toSolutionDirect` \rightarrow Permet de convertir un bag en une solutionDirect (section 2.5).
- `bag_duplicate` \rightarrow Permet de dupliquer un bag sur le tas.

2.5 SolutionDirect - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Le fichier SolutionDirect contient une structure SolutionDirect ayant pour propriétés :

- `instance` \rightarrow Un pointeur vers l'instance associée.
- `itemsTaken` \rightarrow Un tableau de booléens représentant l'état de chaque item (pris ou non).

Les fonctions suivantes s'appliquent à partir d'une structure SolutionDirect :

- `solutionDirect_create` \rightarrow Afin de créer une SolutionDirecte sur le tas (par default aucun item n'est pris).
- `isolutionDirect_destroy` \rightarrow Afin de détruire une SolutionDirecte précédemment créé par `solutionDirect_create`.
- `solutionDirect_evaluate` \rightarrow Fonction score pour une solution directe.
- `solutionDirect_doable` \rightarrow Indique la faisabilité d'une solution directe.
- `solutionDirect_print` \rightarrow Affiche une solution directe dans la console.
- `solutionDirect_saveToFile` \rightarrow Enregistre une solution directe dans un fichier.
- `solutionDirect_takeItem` \rightarrow Marque un item comme pris dans la solution directe.
- `solutionDirect_duplicate` \rightarrow Duplique une solution directe sur le tas.
- `solutionDirect_isItemTaken` \rightarrow Indique si un item est pris dans la solution directe.

2.6 SolutionIndirect - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Le fichier SolutionIndirect contient une structure SolutionIndirect ayant pour propriétés :

- `instance` \rightarrow Un pointeur vers l'instance associée.
- `itemsOrder` \rightarrow L'ordre dans lequel les items seront ajoutés au sac.
- `bag` \rightarrow Un pointeur vers un sac associé à la solution indirecte.

Les fonctions suivantes s'appliquent à partir d'une structure SolutionIndirect :

- `solutionIndirect_create` \rightarrow Afin de créer une SolutionIndirecte sur le tas (par default item order ne contient que des '-1').
- `solutionIndirect_destroy` \rightarrow Afin de détruire une SolutionIndirecte précédemment créé par `solutionIndirect_create`.
- `solutionIndirect_decode` \rightarrow Decode une solution indirecte, permettant de créer le sac associé.

- solutionIndirect_evaluate → Fonction score pour une solution indirecte.
- solutionIndirect_doable → Indique la faisabilité d'une solution indirecte.
- solutionIndirect_print → Affiche une solution indirecte dans la console.
- solutionIndirect_saveToFile → Enregistre une solution indirecte dans un fichier.
- solutionIndirect_getItemIndex → Renvoie l'indice de l'item à la position demandé de la liste de remplissage du sac.
- solutionIndirect_getIndexItem → Renvoie l'indice dans la liste de remplissage d'un item donné.
- solutionIndirect_duplicate → Duplique une solution indirecte sur le tas.

2.7 Solution - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Le fichier Solution contient une structure Solution permettant de rassembler une solution directe ou indirecte afin d'avoir une manipulation plus généraliste de celles-ci. Cette structure a pour propriétés :

- instance → Un pointeur vers l'instance associée.
- type → Une énumération pouvant prendre les valeurs DIRECT ou INDIRECT, représentant le type de la solution contenue.
- solutions → La solution en elle-même, dépendant du type. Union de SolutionDirect et SolutionIndirect.
- solveTime → Un entier représentant le temps de calcul pour obtenir la solution.

Les fonctions suivantes s'appliquent à partir d'une structure SolutionIndirect :

- solution_saveToFile → Enregistre une solution dans un fichier.
- solution_getTimeDiff → Renvoie la différence en secondes entre deux structures timeb.
- solution_evaluate → Fonction score pour une solution.
- solution_doable → Indique la faisabilité d'une solution.
- solution_duplicate → Duplique une solution sur le tas.
- solution_destroy → Afin de détruire une Solution.
- solution_fromIndirect → Crée une solution à partir d'une solution indirecte.
- solution_fromDirect → Crée une solution à partir d'une solution directe.
- solution_full → Renvoie une solution contenant tout les items de l'instance.

2.8 Heuristic - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Dans le cadre des heuristiques, nous avons dû implémenter nos propres critères de sélection. Nous allons ici vous en présenter deux. Les applications de ces fonctions sont présentes dans le fichier scheduler que nous verrons à la section 2.9.

Le premier ("allDimensionsWeighted") se base sur l'algorithme de la dimension critique mais prend cette fois-ci en compte toutes les dimensions. Pour cela nous calculons pour l'item à l'index i un ratio qui est $r_i = \sum_{j=0}^m \frac{w_j}{W_j}$. Ce ratio sert par la suite à calculer un score temporaire afin d'appliquer l'heuristique $score_i = \frac{v_i}{r_i}$. De cette manière, plus l'item remplira le sac, plus le diviseur sera important et par conséquent, l'item aura un score faible.

Le second ("exponential") se base sur le taux de complétion des différentes dimensions ainsi que le poids de l'item. Pour cela, Pour cela nous calculons pour l'item à l'index i un ratio qui est $r_i = \sum_{j=0}^m w_j \times \exp \frac{20 \times Bw_j}{W_j}$ avec Bw_j le poid actuel du sac dans la dimension j . Ce ratio sert par la suite à calculer un score temporaire afin d'appliquer l'heuristique $score_i = \frac{v_i}{r_i}$. De cette manière plus une dimension est pleine, plus elle impliquera un diviseur élevé. De plus, plus un item prend une place importante dans l'une des dimensions, plus le diviseur sera élevé. L'exponentielle permet de rapidement élever le diviseur si l'item prend trop de place et "efface" les différence entre les poids de chaque items afin de préléger les items qui remplissent peu le sac.

Les fonctions contenues dans Heuristic sont :

- heuristic_search → Lance la recherche heuristique.
- heuristic_getList → Renvoi la liste des items ordonnées en fonction du scheduler choisi.

2.9 Scheduler - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Les fonctions générales de ce fichier sont :

- scheduler_removeFromList → Pour retirer un indice d'item d'une liste tout en le retournant.
- scheduler_appendToList → Pour ajouter un indice d'item à une liste.
- scheduler_sortArray → Pour trier une liste d'index d'items à partir d'une liste de score.

Les fonctions restantes représentent les différentes méthodes de tri pour l'heuristique, celles-ci sont de la forme : scheduler_[Name] étant la fonction renvoyant la liste d'index d'items ordonnée, scheduler_[Name]_score calculant le score d'un item. Les différents noms sont :

- random → Créant une liste aléatoirement ordonnée.
- value → Créant une liste ordonnée selon la valeur des items.
- allDimensions → Créant une liste grace au ratio $\frac{v_i}{\sum_{j=0}^m w_{i,j}}$.
- forDimension → Créant une liste à partir du ratio valeur / poids dans une dimension donnée (utilisé principalement pour la dimension critique).
- allDimensionsWeighted → Expliqué à la section 2.8.
- exponential → Expliqué à la section 2.8.

2.10 MetaheuristicLocal - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Ce fichier comporte toutes les fonctions associées à la recherche locale :

- metaheuristicLocal_search → Permet de lancer la recherche locale. Les paramètres ajoutés sont :
 - solutionType → Le type de la solution à créer (sirecte ou indirecte).
 - operatorSearch → L'opérateur de recherche. Pour le moment seulement 0 est utilisé dans le cas du direct et de l'indirect. Cela correspond à la ajouter et inverser pour le directe et inverser pour l'indirecte.

- `metaheuristicLocal_getNeighbours` → Créé une liste de voisins d'une solution basé sur l'opérateur de recherche choisi.
- `metaheuristicLocal_swapItem` → Créé une liste de voisins d'une solution basé sur l'opérateur de recherche d'inversion (indirecte).
- `metaheuristicLocal_addItem` → Créé une liste de voisins d'une solution basé sur l'opérateur de recherche d'ajout (directe).
- `metaheuristicLocal_invertItem` → Créé une liste de voisins d'une solution basé sur l'opérateur de recherche d'inversion (directe).
- `metaheuristicLocal_addAndInvertItem` → Créé une liste de voisins d'une solution basé sur la concaténation des opérateurs de recherche d'ajout et d'inversion (directe).

2.11 MetaheuristicTabou - Codé à 100% | Testé à 99.9% | Fonctionne: Oui | Valgrind: OK

Ce fichier comporte une première structure `Movement` permettant d'identifier un mouvement $a \Leftrightarrow b$. Ses propriétés sont :

- `a` → L'index du premier élément à inverser.
- `b` → L'index du second élément à inverser.

Les fonctions suivantes lui sont associées :

- `movement_equals` → Permet de vérifier si deux mouvements sont les mêmes.
- `movement_applyMovement` → Applique le mouvement sur une solution.
- `movement_duplicate` → Duplique un mouvement sur le tas.

Une deuxième structure `Tabou` représente la liste des mouvements interdits et à pour propriétés :

- `movements` → Une liste de pointeurs de mouvements.
- `size` → La taille actuelle de la liste.
- `changes` → Le nombre d'ajouts à la structure (pour pouvoir écrire de manière "continue" dans la liste).
- `max` → La taille maximum de la liste.

Les fonctions suivantes lui sont associées :

- `tabou_create` → Permet de créer une structure `Tabou` sur le tas.
- `tabou_appendMovement` → Permet d'ajouter un mouvement à la liste `tabou`.
- `tabou_isMovementTabou` → Indique si un mouvement est présent dans la liste `tabou`.
- `tabou_destroy` → Détruit une structure `tabou` précédemment créée par `tabou_create`.

Les fonctions suivantes concernent la recherche `tabou` :

- `metaheuristicTabou_search` → Permet de lancer la recherche `tabou`. Les paramètres ajoutés sont :
 - `solutionType` → Le type de la solution à créer (directe ou indirecte).⁴
 - `iterationMax` → Le nombre maximum d'itérations à effectuer lorsque nous trouvons la même solution.
 - `tabouMax` → La taille maximum de la liste `tabou`.

- aspiration → Un booléen représentant l'utilisation ou non de l'aspiration.
- metaheuristicTabou_getNeighbourFromMovement → Duplique une solution puis lui applique un mouvement.
- metaheuristicTabou_getMovements → Renvoi la liste de toutes les permutations possibles.

2.12 MetaheuristicGenetic - Codé à 100% | Testé à 99.9% | Fonctionne: ?
| Valgrind: OK

2.13 MetaheuristicKaguya - Codé à 100% | Testé à 0% | Fonctionne: ? |
Valgrind: ?

Conclusion

Projet tutoré 2 : Sac à dos

Rapport de projet S4

Résumé : Projet ayant pour objectif la réalisation d'un algorithme cherchant des solutions au problème du sac à dos multidimensionnel.

Mots clé : sac à dos, algorithme, C, heuristique, metaheuristique, parser, directe, indirecte

Abstract : Project which objective is to find solutions for the multidimensional Knapsack problem.

Keywords : backpack, Knapsack, algorithm, C, heuristic, metaheuristic, parser, direct, indirect

Auteur(s)

Thomas Couchoud

[thomas.couchoud@etu.univ-tours.fr]

Victor Coleau

[victor.coleau@etu.univ-tours.fr]

Encadrant(s)

Yannick Kergosien

[yannick.kergosien@univ-tours.fr]

Polytech Tours

Département DI

Ce document a été formaté selon le format EPUProjetPeiP.cls (N. Monmarché)

École Polytechnique de l'Université de Tours
64 Avenue Jean Portalis, 37200 Tours, France
<http://www.polytech.univ-tours.fr>