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

Encadrant(s)

Yannick Kergosien
[yannick.kergosien@univ-tours.fr]
Polytech Tours
Département DI

Version du 27 janvier 2017

Table des matières

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 doublons. 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 nommées *population_ create(...)* ou encore *metaheuristicGenetic_ search(...)*.

Concernant l'organisation des fichiers en eux-mêmes, ils sont regroupés par types localisés à des endroits différents. En effet, nous avions commencé par mettre tous nos .c et .h dans un même dossier. Cependant, le projet grandissant assez vite, il est rapidement arrivé un stade où l'on se perd. Pour éviter cela, nous avons séparé 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 méta-heuristique locale.

— MetaheuristiqueTabou ⟶ Regroupant les fonctions liées aux différents algorithmes pour la méta-heuristique tabou et contenant la structure Tabou.
— MetaheuristiqueGenetic ⟶ Regroupant les fonctions liées aux différents algorithmes pour la méta-heuristique génétique et contenant la structure Population.
— MetaheuristicKaguya ⟶ Regroupant les fonctions liées à notre méta-heuristique personnelle (contenant les structures Clan, ClanMember et DNA).
— MetaheuristicKaguyaTemp ⟶ Regroupant les fonctions liées à notre méta-heuristique personnelle (contenant les structures Clan, ClanMember et DNA) en phase de reconception.

1.3 Macros

Vous pourrez trouver tout au long du projet des macros MMALLOC et RREALLOC. Celles-ci permettent l'allocation (ou réallocation) sur le tas d'une variable puis de vérifier cette allocation. La définition de cette macro est présente dans utils.h. Les paramètres sont :

— Le pointer à malloc/realloc.
— Le type à allouer.
— La quantité de ce type à allouer.
— Un message d'erreur en cas d'échec.

Chapitre 2

Explication des fichiers

Nous allons maintenant décrire brièvement les différentes fonctions contenues 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 devions 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 avions 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 réaliser. Puis, rapidement, nous avons implémenté la seconde méthode. Celle-ci nous parut plus adéquate pour nos utilisations car elle permet d'éviter une consommation importante de mémoire inutilement. 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 ses 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 coté 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 ficher.

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

Le fichier Instance comporte une structure nommée Instance qui contient les 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'entiers 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 appelée. Le tableau d'items est créé mais chaque item devra être initialisé grâce à item_setWeight.
— instance_getItem ⟶ Permet de récupérer une 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 ⟶ Ajoute du poids dans le sac dans la dimension donnée.
— bag_saveItems ⟵ Ecrit le bag dans un fichier.
— bag_print ⟶ Affiche le bag dans la console.
— bag_getCriticDimension ⟶ Renvoie l'index de la dimension critique.
— bag_toSolutionDirect ⟶ Permet de convertir un bag en une solutionDirect (section 2.5).
— bag_duplicate ⟶ 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 ⟶ Un pointeur vers l'instance associée.
— itemsTaken ⟶ 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 ⟶ Afin de créer une SolutionDirecte sur le tas (par défaut aucun item n'est pris).
— isolutionDirect_destroy ⟶ Afin de détruire une SolutionDirecte précédemment créée par solutionDirect_create.
— solutionDirect_evaluate ⟶ Fonction score pour une solution directe.
— solutionDirect_doable ⟶ Indique la faisabilité d'une solution directe.
— solutionDirect_print ⟶ Affiche une solution directe dans la console.
— solutionDirect_saveToFile ⟶ Enregistre une solution directe dans un fichier.
— solutionDirect_takeItem ⟶ Marque un item comme pris dans la solution directe.
— solutionDirect_duplicate ⟶ Duplique une solution directe sur le tas.
— solutionDirect_isItemTaken ⟶ 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 ⟶ Un pointeur vers l'instance associée.
— itemsOrder ⟶ L'ordre dans lequel les items seront ajoutés au sac.
— bag ⟶ Un pointeur vers un sac associé à la solution indirecte.

Les fonctions suivantes s'appliquent à partir d'une structure SolutionIndirect :
— solutionIndirect_create ⟶ Afin de créer une SolutionIndirecte sur le tas (par default item order ne contient que des '-1').
— solutionIndirect_destroy ⟶ Afin de détruire une SolutionIndirecte précédemment créé par solutionIndirect_create.
— solutionIndirect_decode ⟶ Décode 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 demandée 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 générale 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 ⟶ Détruit une Solution.
— solution_fromIndirect ⟶ Créé une solution à partir d'une solution indirecte.
— solution_fromDirect ⟶ Créé une solution à partir d'une solution directe.
— solution_full ⟶ Renvoie une solution contenant tous 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 du implémenter nos propres critères de sélection. Nous allons ici vous en présenter deux. Les applications de ces fonctions sont présentes dans le ficher 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 est un score temporaire afin par la suite 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 en question prend beaucoup trop de place et "efface" les différences entre les poids de chaque items afin de privilégier les items qui remplissent peu le sac.

Les fonctions contenues dans Heuristic sont :
- heuristic_search ⟶ Lance la recherche heuristique.
- heuristic_getList ⟶ Renvoie 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 représentes 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 ordonnée aléatoirement.
- value ⟶ Créant une liste ordonnée selon la valeur des items.
- allDimensions ⟶ Créant une liste grâce 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).
- allDimentionsWeighted ⟶ 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 (directe ou indirecte).
  — operatorSearch ⟶ L'opérateur de recherche. Pour le moment seul 0 est utilisé dans le cas du direct et de l'indirect. Cela correspond à ajouter puis inverser pour le type direct et inverser pour l'indirect.
— 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 movement à la liste tabou.
— tabou_isMovementTabou ⟶ Indique si un movement est présent dans la liste tabou.
— tabou_destroy ⟶ Détruit une structure tabou précédemment créée par tabou_create.

Les fonctions suivantes concernent la recherche taboue :

— metaheuristicTabou_search ⟶ Permet de lancer la recherche tabou. Les paramètres ajoutés sont :
  — solutionType ⟶ Le type de la solution à créer (directe ou indirecte).‘
  — iterationMax ⟶ Le nombre maximum d'itérations à effectuer lorsque nous trouvons la même solution.
  — tabouMax ⟶ La taille maximum de la liste taboue.
  — aspiration ⟶ Un booléen représentant l'utilisation ou non de l'aspiration.
— metaheuristicTabou_getNeighbourFromMovement ⟶ Duplique une solution puis lui applique un mouvement.
— metaheuristicTabou_getMovements ⟶ Renvoie la liste de toutes les permutations possibles.

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

Ce fichier comporte une structure Population permettant de regrouper plusieurs solution en tant que "génération". Celle-ci contient les propriétées suivantes :
— people ⟶ Une liste de pointeur sur les membres de cette population.
— maxSize ⟶ La taille maximum de la population.
— size ⟶ La taille actuelle de la population.

Les fonctions suivantes lui sont associées :
— population_create ⟶ Permet de créer une population sur le tas.
— population_destroy ⟶ Détruit une population pécédemment créée par population_create.
— population_append ⟶ Ajoute une solution dans la population si il reste de la place.
— population_getBest ⟶ Renvoie la meilleure solution de la population.
— population_getWorst ⟶ Renvoie la pire solution de la population.
— population_duplicate ⟶ Duplique une solution sur le tas.
— population_replace ⟶ Remplace une solution par une autre dans une population.
— population_remove ⟶ Retire une solution d'une population.
— population_evaluate ⟶ Calcule le score de la population (somme des scores des solutions).

Les fonctions de la méta-heuristique sont :
— metaheuristicGenetic_search ⟶ Permet de lancer la recherche génétique. Les paramètres ajoutés sont :
  — solutionType ⟶ Le type de la solution à créer (directe ou indirecte).‘
  — populationSize ⟶ La taille des populations.
  — mutationProbability ⟶ La Probabilité de mutation.
  — maxIterations ⟶ Le nombre de générations à créer.
  — styleNaturalSelection ⟶ Le style de sélection naturelle à utiliser.
  — styleParentSelection ⟶ Le style de selection des parents à utiliser.
— metaheuristicGenetic_selectParents ⟶ Sélectionne le mode d'élection des parents parmi les méthodes "selectParents" suivantes.

— metaheuristicGenetic_selectParentsFight ⟶ Sélectionne les deux parents selon une méthode de combat. Deux paires de combattants sont choisis aléatoirement dans la population. Les meilleurs de chaque paire sont désignés comme parents.
— metaheuristicGenetic_selectParentsRoulette ⟶ Sélectionne les deux parents selon une méthode totalement aléatoire. Chaque individu a une probabilité d'être sélectionné égal au pourcentage de sa valeur parmi la valeur totale de la population.
— metaheuristicGenetic_breedChildren ⟶ Sélectionne le mode de création des enfants parmi les méthodes breedChildren" suivantes.
— metaheuristicGenetic_breedChildrenPMX ⟶ Créé deux enfants à partir des deux parents selon la méthode PMX (uniquement pour les solutions indirectes).
— metaheuristicGenetic_breedChildren1Point ⟶ Créé deux enfants à partir des deux parents selon la méthode de croisement en 1 point (uniquement pour les solutions directes).
— metaheuristicGenetic_mutation ⟶ Permet de faire muter une solution.
— metaheuristicGenetic_naturalSelection ⟶ Sélectionne le mode de remplacement de l'ancienne génération par la nouvelle parmi les méthodes "naturalSelection" suivantes.
— metaheuristicGenetic_naturalSelectionGeneration ⟶ Remplace simplement la totalité de l'ancienne génération par la nouvelle.
— metaheuristicGenetic_naturalSelectionElitist ⟶ Remplace l'ancienne génération par les meilleurs individus des ancienne et nouvelle populations réunies.
— metaheuristicGenetic_naturalSelectionBalanced ⟶ Remplace l'ancienne génération par les 50% meilleurs individus de l'ancienne génération les 50% meilleurs de la nouvelle.

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

A la suite des méta-heuristiques présentées lors des TDs, il nous a été demandé de proposer notre propre méta-heuristiques. Nous avons nommé celle-ci Kague (metaheuristicKaguya), du plus ancien conte japonnais retrouvé. Celle-ci part d'une solution où tous les items sont présents, puis tente d'en retirer certain. Nous essayons de traiter tous les cas possibles mais cela demande énormément de ressource. Pour palier à cette contrainte, deux optimisations sont apportées : une nouvelle structure ClanMember représente un solution de manière simplifiée, économisant de l'espace mémoire, et nous stoppons le processus de génération pour une solution dès que celle-ci est réalisable.

Cette méta-heuristique peut être intéressante pour les sacs contenant beaucoup d'items dans leur solution optimale, mais s'avère dangereuse si ce n'est pas le cas. Nous avons tenté plusieurs tests cependant, le temps de résolution et la place prise en mémoire excessivement importants du au grand nombre de cas explorés nous ont empêché d'avancer plus loin avec cette méta-heuristique telle qu'elle. [Un début de résolution de ce problème a été ajouté mais, bien qu'optimisant l'espace mémoire, ne diminue pas le temps d'exécution]

La cause des soucis précédemment évoqués est la présence de doublons parmi nos solutions. En effet, un sac plein auquel on enlèverait les items 2 et 3 est identique à un sac privé des items 3 et 2. Afin de palier à ce problème de duplicatas, nous avons pensé à créer une table de hashage qui permettrait de chercher et d'éliminer les duplicatas de manière plus efficace (parcourir une liste de 900 000 solutions est toujours mieux qu'une de 90 000 000). Malheureusement, ce

système ne put être achevé dans le temps imparti bien qu'une ébauche puisse être trouvée dans les fichiers metaheuristiqueKaguyaTemp.

Parlons maintenant du fichier metaheuristiqueKaguya en lui-même : deux structures sont présentes. La première est ClanMember et représente une séquence d'items à retirer afin d'effectuer une solution. Ses propriétés sont :

— DNA ⟶ Un tableau d'int représentant les items à retirer.
— dilution ⟶ Le nombre d'items à retirer (indiquant dans le même temps de la génération du membre).

Les fonctions lui étant associées sont :

— clanMember_ancestor ⟶ Créé un "membre originel" dont l'ADN est vide, représentant donc une solution où tous les items de l'instance seraient pris dans le sac.
— clanMember_destroy ⟶ Détruit un membre.
— clanMember_generation ⟶ Créé tous leurs enfants du membre en ajoutant à l'ADN de celui-ci un des items encore non enlevés.
— clanMember_duplicate ⟶ Duplique un membre sur le tas.
— clanMember_doable ⟶ Vérifie si un membre représente une solution réalisable.
— clanMember_toSolution ⟶ Crée une solution à partir d'un membre.
— clanMember_evaluate ⟶ Fonction objectif appliqué à un membre. En réalité, celle-ci commence par créer la solution associée, l'évalue puis la détruit.
— cleanMember_equals ⟶ Vérifie l'égalité de deux membres (Retirer les objets 2 et 3 revient au même que de retirer les objets 3 et 2).

La deuxième est Clan et représente un ensemble de plusieurs membres. Ses propriétés sont :

— instance ⟶ Un pointeur sur l'instance concernée.
— type ⟶ Le type de solution à créer (seul DIRECT est possible pour le moment).
— size ⟶ Le nombre de membres dans le clan.
— people ⟶ Une liste de pointeurs sur les différents membres.

Les fonctions lui étant associées sont :

— clan_create ⟶ Crée un clan sur le tas.
— clan_append ⟶ Ajoute un membre à un clan.
— clan_remove ⟶ Retire un membre d'un clan.
— clan_generation ⟶ Génère les enfants de chacun des membres du clan.
— clan_dispertion ⟶ Trouve les membres représentant des solutions réalisables et les envoie dans le clan réservé aux solutions faisables afin de ne pas générer d'enfants moins bons que le parent.
— clan_extinction ⟶ Détruit un clan et renvoie le meilleur des membres.

La seule fonction de la méta-heuristique pour le moment est metaheuristicKaguya_search et permet de lancer la recherche. Le seul paramètre pour le moment est le type de solution à créer (seul DIRECT est supporté pour le moment).

Conclusion

Ce projet fut une bonne et très intéressante expérience. Le problème en lui-même est assez concret, ce qui permet d'en assimiler les enjeux et difficultés rapidement.

Nous avons eu un départ assez rapide et avons trouvé une heuristique qui avait d'assez bons résultats. Cependant nous avons ralenti sur la fin ce qui nous a empêché de faire tourner l'algorithme sur de vrai données afin d'interpréter les résultats. De plus la méta-heuristique inventée est probablement à revoir car elle explore énormément de résultats et prend par conséquent un temps d'exécution trop important.

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 multidimentional Knapsack problem.

Keywords : backpack, Knapsack, algorithm, C, heurictic, 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