Problème du sac-à-dos

Résolution par un algorithme Branch & Bound

1. Architecture du programme

- /sacs — contient cinq instances de test

- /src
  - algo.c/h — algorithme branch-and-bound
  - extern.h — variables globales utilisées
  - main.c — utilisation de l'exécutable
  - queue.c/h — définition de la file de priorité, implémentée par un tas
  - util.c/h — fonctions d'affichage et de lecture d'une instance de test

- Makefile — instructions de compilation

2. Utilisation du programme

- make — compilation
- make clean — nettoyage des fichiers .o et de l'exécutable

- ./main sacs/x [options]
  - -v — flag de débogage
  - -h — afficher le message d'erreur

3. Implémentation du Branch & Bound

Nous avons préféré implémenter l'algorithme dans sa version itérative plutôt que sa version récursive.

Ainsi, nous simulons les appels récursifs par une pile. Ce choix nous permet alors d'utiliser très facilement plusieurs méthodes de parcours de l'arbre :

- en profondeur d'abord (depth-first search)
- en largeur d'abord (breadth-first search)
- le plus prometteur d'abord (best-first search)

GOUDOUT Lénaïc - DEBERNARDI Hippolyte

La fonction secondaire, *upper_bound(Node n, Object *objects)*, nous permet de connaître le profit fractionnaire maximal pour le noeud n.

La fonction principale de l'algorithme, *knapsack(Object *objects)*, fonctionne donc en empilant les deux choix possibles pour le noeud à une profondeur donnée.

4. Exploration de l'arbre de recherche

Naïvement, nous avons commencé par implémenter la recherche la plus simple à savoir l'exploration en largeur d'abord. En effet, empiler simplement les deux noeuds possibles à une profondeur p est un jeu d'enfant : on ajoute en bas de la pile le noeud où l'on prend l'objet en premier puis encore en dessous le noeud où l'on ne prend pas l'objet.

Il est évident que cette solution, acceptable pour des tailles de sac et un nombre d'objets faibles, devient rapidement inexploitable (le cinquième sac n'a pas pu être résolu).

Nous choisissons alors d'implémenter une file de priorité, elle même implémentée par un tas. Ce choix nous garantit les complexités en temps suivantes :

- dequeue en O(log n)
- enqueue en O(log n)
- is_empty en O(1)

où n est le nombre de noeuds dans la file.

5. Annexes

La case en haut à gauche indique le nombre d'objets puis la capacité du sac.

(15, 1000)	Breadth-first search	Best-first search	Breadth/Best ratio
Noeuds visités	66	25	2.64
Nombre d'objets visités dans la fonction upper_bound	346	47	7.36
Temps d'exécution (en ms.)	0.72	0.132	5.45

(20, 2000)	Breadth-first search	Best-first search	Breadth/Best ratio
Noeuds visités	18056	223	80.97
Nombre d'objets visités dans la fonction upper_bound	184803	459	402.62
Temps d'exécution (en ms.)	15.929	0.361	44.12

(40, 2000)	Breadth-first search	Best-first search	Breadth/Best ratio
Noeuds visités	19559	2359	8.29
Nombre d'objets visités dans la fonction upper_bound	195326	1302	150
Temps d'exécution (en ms.)	17.748	1.948	9.10

(100, 2000)	Breadth-first search	Best-first search	Breadth/Best ratio
Noeuds visités	138859	5497	25.26
Nombre d'objets visités dans la fonction upper_bound	1548446	1455	1064
Temps d'exécution (en ms.)	63.788	4.531	14.08

(500, 2000)	Breadth-first search	Best-first search	Breadth/Best ratio
Noeuds visités	2794030	2377	1175
Nombre d'objets visités dans la fonction upper_bound	35218458	1234	28540
Temps d'exécution (en ms.)	851.606	2.473	344.36

(1000, 20000)	Breadth-first search	Best-first search	Breadth/Best ratio
Noeuds visités	-	597	-
Nombre d'objets visités dans la fonction upper_bound	-	48337	-
Temps d'exécution (en ms.)	-	4.304	-

N.B. : Sur un ordinateur équipé d'un processeur Intel® Core™ i5-7300HQ CPU @ 2.50GHz × 4, l'algorithme en breadth-first search est arrêté par le système après plus d'une dizaine de minutes, sans donner de résultat.