Rapport du projet de Recherche Opérationnelle

NOBLET Yvan, GIBAUD Lary

Table des matières

1 Introduction

Ceci est le rapport du projet de recherche opérationnelle : *Une méthode simple pour la résolution exacte d'un problème de tournée de véhicule avec capacité.*

Le but est de fournir un programme permettant de définir un plan de livraison de clients par des drones. Ce but est atteint par un programme écrit en langage C et l'utilisation de la bibliothèque GLPK.

2 Analyse

2.1 Squelette du programme

Le squelette du programme C nous est donné. Celui-ci contient les inclusions de librairies nécessaires pour faire fonctionner GLPK ainsi que le parser permettant de lire les fichiers de données.

2.2 Fichiers de données

Deux jeux de données nous sont fournis (les répertoires A et B) ainsi que l'exemple de l'énoncé (exemple.dat). Le format du fichier de données est ainsi :

```
ligne 1 est : nombre de clients
ligne 2 est : capacité d'un drone
les lignes suivantes sont : le distancier
```

2.3 Formalisation du problème

On nous donne :

- Ω, un ensemble de numéros de clients, $\Omega = [\![1\,;n]\!]$.
- Δ, une matrice $n \times n$ telle que le nombre (entier) à l'intersection de la ligne i et de la colonne j représente le coût ou la distance pour aller du client i au client j.

- *Dem*, une fonction donnant la demande (volume ou poids) de chaque client.
- *Cap*, la capacité des drones (tous les drones ont la même capacité, pour une instance donnée).

On nous demande d'écrire un programme effectuant trois étapes :

1. Enumérer les regroupements possibles de clients. Ce sont tous les ensembles de clients dont la somme des demandes est inférieure à la capacité d'un drone :
$$\theta = \{\ a \subseteq \Omega \mid \Sigma_{x \in a}(Dem[x]) \leq Cap\ \}$$

2. Pour chacun de ces regroupements, déterminer la plus courte tournée partant du dépôt, visitant une et une seule fois chaque client, et revenant au dépôt. Cela revient à trouver, pour tout regroupement, le chemin le plus court partant du dépot(client 0), passant par tous les clients du regroupement et retournant au dépot.

   Soit $a \in \theta, |a| = t$.
   $\big(bc(a)^{\,1} = (a_0, a_1, \ldots, a_t, a_{t+1})\big) \iff$
   $\big(\forall i, j \in [\![1\,;t]\!], \quad a_i, a_j \in a \wedge (i \neq j \implies a_i \neq a_j) \quad \wedge$
   $a_0 = a_{t+1} = 0 \quad \wedge$
   $\sum_{i=0}^{t} \Delta[a_i, a_{i+1}] = min_{(b_1,\ldots,b_t)\text{permutation des } (a_1,\ldots,a_t)} \sum_{i=0}^{t} \Delta[b_i, b_{i+1}], b_0 = b_{t+1} = 0\big)$.

3. En déduire une instance de problème de partitionnement d'ensemble, puis la résoudre. Ayant, pour chacun des regroupements, le coût minimum associé, il ne reste qu'à trouver la combinaison de regroupements permettant de livrer tous les clients et assurant une distance parcourut minimale. Ceci est formalisé dans le sujet du projet.

3 Choix tactiques et techniques

3.1 Organisation des fichiers

Le programme est découpé en trois fichiers :

globals.c/globals.h contient :
- Des directives préprocesseur *#define* pour pouvoir compiler en mode débuggage.
- Les définitions de deux structures de données et les fonctions/procédures associées : *List* et *Base*.

fonctions.c/fonctions.h contient :
- Les fonctions/procédures pour résoudre l'étape 1.
- Les fonctions/procédures pour résoudre l'étape 2.

strategy.c contient une structure de données permettant de choisir entre deux algorithmes à l'exécution pour résoudre l'étape 2.

projet_NOBLET_GIBAUD.c contient le squelette rempli.

script.sh script bash permettant de lancer le programme (ses deux versions) sur tous les jeux de données, et de stocker les résultats dans un répertoire approprié.

3.2 Organisation du code et des données

3.2.1 Introduction

Nous avons décidé de ne faire qu'une description minimale des algorithmes écrits, parce que ce serait comme les réécrires. Le code source commenté est à votre disposition.

3.2.2 Choix des structures de données

Comme annoncé ci-dessus, nous utilisons des listes [2]. Nous avons choisi cette structure de données pour sa simplicité ainsi que son efficacité en termes de complexité temporelle [3] pour :

- l'ajout en tête,
- la suppression en tête.

Nous avons opté pour des listes de `(void*)`, cela nous permet d'avoir des listes d'entier, de listes, d'entiers et de listes, ...

Nous utilisons également des tableaux C...

3.2.3 Enumération des regroupements

Nous avions tout d'abord pensé l'algorithme récursivement, puis nous l'avons "dérécursifié" par un "tant que" et l'ajout d'une pile [4]. Afin de rester clairs et concis, nous allons expliquer les algorithmes, quand ce sera possible et quand il ont été pensé de cette manière, de façon récursive, même si nous les avons transformé en algorithmes de type impératif pour des raisons de performance [5].

```
List* enum_regroups(int* demands, int nbclients, int capacity);
```

Cette fonction commence par effectuer [6] un tri linéaire (tri casier) des clients par rapport à la taille de leur demande. Informellement, voilà ce qu'il se passe :

On dispose de deux variables importantes : *base* et *rest*. *base* stocke un ensemble de clients, ainsi que la somme de leurs demandes, *rest* est un ensemble de clients [7] triés par demande croissante à tester (ie : est-ce que sa demande sommée à la demande de *base* dépasse la capacité d'un drone ?).

Plusieurs cas se présentent :

1. *rest* est vide. Dans ce cas c'est qu'on est à une feuille du parcours de notre arbre [8].
2. *rest* n'est pas vide. Dans ce cas on a une liste de clients potentiellement ajoutable à la base pour constituer de nouveaux regroupements. On parcourt donc les sous-listes de *rest*, tant que la tête est un client que l'on peut ajouter, on appel récursivement *enum_regroups* avec la nouvelle base, et la queue de la sous-liste actuelle de *rest*.

2. Voir *globals.c* ainsi que la note 1.
3. temps constant.
4. *stack*, voir *fonction.c*, une pile est une bête liste.
5. le C ne gérant pas la récursion terminale, entre autres.
6. Appel de counting_sort.
7. Ces clients sont des clients n'apparaissant pas dans *base*.
8. L'algorithme peut être vu comme un parcours en profondeur d'abord d'un arbre n-aire, n étant le nombre de clients.

Deux choses sont essentielles ici :

(a) Quand un client n'est pas ajoutable, les clients suivants non plus (clients ordonnés par leur demande).

(b) Quand un client est ajoutable, le reste avec lequel on fait un appel récursif n'est pas *rest* privée du client mais la queue de la liste dont le client est la tête. Cela garantit l'unicité des regroupements générés.

De la gestion de la mémoire. Lorsque nous ajoutons une nouvelle base et (ce qui est la même chose) un nouveau regroupement, nous ne faisons pas une copie de la base, nous ajoutons simplement en tête de la base le nouveau client. Etant donnés :

(a) L'ajout de nouveaux regroupements se fait dans une liste utilisée comme une pile.

(b) La structure spécifique de la constitution des regroupements.

(c) Nous avons besoin de parcourir la liste des regroupements[9] pour effectuer la phase 2.

Il nous est simple de libérer la mémoire, de façon symétrique à son allocation.

3.2.4 Enumeration des tournées

Nous avons implémentés deux algorithmes différents.

Algorithme 1 : Johnson-Trotter. Le premier algorithme est une implémentation de l'algorithme d'énumération complète des permutations Johnson-Trotter. On utilise une paire de tableaux, on définit la structure de données nécessaire au fonctionnement de l'algorithme[10].

Algorithme 2 : Algorithme maison. La structure du second algorithme est très similaire à la structure de l'algorithme d'énumération des regroupements. On utilise le fait de se trouver dans un repère euclidien et donc de disposer de l'inégalité triangulaire pour "couper", dès lors qu'on a un majorant, si la distance du chemin actuel (non complet, ie : tous les clients du regroupement n'ont pas encore été visités) + la distance du dernier visité au dépot dépasse le majorant.

Conclusion. L'algorithme 2 offre un certain intérêt pour les jeux de données plus difficiles puisqu'il "coupe".

4 Idées d'amélioration

Utiliser des algorithmes dédiés au problème du voyageur de commerce, plus efficaces qu'une énumérations (algorithme de Little, si on veut rester dans l'optimalité). Utilisation d'heuristiques, choisir d'accepter une solution approchée dans des cas trop difficiles. Ne pas énumérer tous les regroupements possible, heuristique "plus proches voisins". Pour le voyageur de commerce, heuristique

9. ou de depiler entièrement.
10. paire entier * direction

"insertion la moins chère", on est dans un repère euclidien. Une petite journée de recherche des heuristiques existantes et des dizaines d'implémentations seraient possibles.

5 Conclusion

Finalement, on se rend compte que la façon de résoudre le problème n'est pas la bonne. On ne peut résoudre les instances trop difficiles. Chacune des trois phases prend effectivement un temps trop important [11].

11. voir le dossier logs