Recherche Opérationnelle
Rapport de Projet


# BladeFlyer II : conquest of water

BLAISONNEAU Romain
CHARPENTIER Antoine

Groupe : 601 A

2016-2017

# Table des matières

# Introduction

Constitué d'une composante de programmation C(++) autant que d'une mise en application des connaissances en recherche opérationnelle, le sujet étant relativement approché du réel, ce projet de fin d'année scolaire fut une bonne occasion de consolider
nos connaissances et de tester notre capacité de raisonnement abstrait tout en continuant de penser efficacement.

De surcroît, la nécessité de pouvoir gérer de grosses données efficacement a donné lieu à une sorte de compétition au sein de la promotion, ce qui mène à l'entraide et au partage, ce qui fut franchement agréable durant la réalisation du code.

# Choix d'implémentation

## C/C++

Il était originalement question de faire le projet en C pur, par entêtement. Cependant, de par le choix que nous avons fait de calculer tous les sous-circuits réalisables avant de résoudre le problème de partitionnement d'ensemble, et à fortiori de les stocker quelque part, l'utilisation d'une structure de données dynamique était obligatoire. Nous avons initialement utilisé un tableau statiquement alloué, ce qui nous forçait à prévoir la taille dans le pire des cas, c'est-à-dire où tous les sous circuits sont réalisables, et ainsi à allouer une structure d'une taille valant $2^n$ (nombre de parties d'un ensemble) ou n était le nombre de sources à considérer. On peut se douter qu'allouer un tableau de $2^{50}$ * sizeof(int) n'est pas possible.
Nous avons donc opté pour le C++ dans l'unique but de pouvoir utiliser un unique `vector` dans lequel on stocke les données des sous circuits réalisables

## Résolution du "traveling salesman" algorithmiquement ou avec glpk

Nous avons décidé d'utiliser glpk pour trouver le plus court chemin réalisable pour chaque sous-ensemble de sources, plutôt que de le faire algorithmiquement. La résolution non "brute force" de ce problème est en effet réputée pour être fortement problématique. Etant donné que la complexité de la résolution brute force est en $n!$ , et que la complexité factorielle est le cauchemar du programmeur, nous nous sommes instinctivement dit que glpk serait plus performant, ce qui n'est en définitive pas si évident, vu les coûts entraînés par la mise en place des contraintes / variables. De plus, on n'est pas du tout assurés de l'efficacité miraculeuse de glpk face à un problème en variables entières, quand bien même il est raisonnable de penser qu'il agit de manière plus intelligente qu'en utilisant le brute force.

## Gestion des listes, structures

Etant donné que la transition entre C et C++ s'est faite un peu tardivement dans l'avancement du projet, toutes les listes sauf celle qui posait problème sont gérées par des structures qui contiennent leur nombre d'éléments. On dispose ainsi des structures :

`donnees_map` qui contient les données du problème : nombre de sources, la capacité du drone, la liste des quantités d'eau pour les sources, et la matrice des distances.
`donnees_circuit` qui est une liste de sommets associée à son nombre d'éléments, ainsi qu'à la longueur de son trajet optimal une fois qu'il est calculé.
`liste_de_donnees_circuits` qui est une liste de donnees_circuit associée à son nombre d'éléments.
`sous_circuit` qui est une liste de sommets associée à son nombre d"éléments
`liste_de_sous_circuits` qui est une liste de sous_circuit associée à son nombre d'éléments.

# Détails des fonctions

## Lecture du fichier de données

```
void remplir_donnes_map(char *nomfich, donnees_map *don);
```

Cette fonction est, comme on peut s'y attendre, relativement simple. Elle lit le nombre n de sources, puis la charge du drone, puis n quantités d'eau qu'elle place dans la liste correspondante, puis n * n éléments qu'elle place dans le tableau à deux dimensions des distances.

## Calcul des sous circuits et de leur longueur associée

```
void remplir_liste_donnees_circuits(
liste_de_donnees_circuits *list_donnees,
donnees_map map,
int indice,
donnees_circuit sous_circ,
int sterile_call);
```

Cette fonction effectue le remplissage de la structure contenant les sous circuits réalisables ainsi que leur taille de trajet optimisée. On passe en paramètre un pointeur sur la liste que l'on veut remplir, les données du problème, et un indice, un sous circuit et un flag entier "sterile_call" qui seront utilisé pour la récursion.

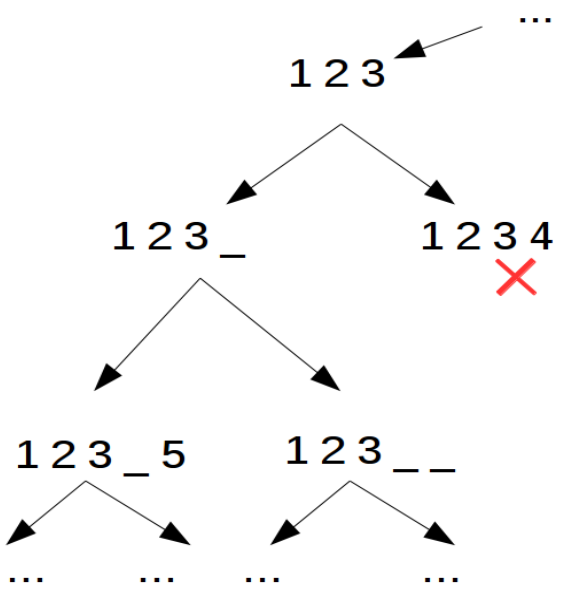
`indice` représente le sommet que l'on "regarde" à l'appel courant, c'est à dire que l'on va brancher sur deux appels récursifs : l'un l'ajoutant au circuit que l'on génère et pas l'autre.

`sous_circ` est le sous circuit que l'on modifie pour générer tous les sous-circuits réalisables

`sterile_call` est un indicateur qui dénote le fait que l'appel parent à ou non ajouté son sommet. Si le circuit est réalisable et que sterile_call est à 0 (=non), on se trouve en présence d'un circuit réalisable qui n'a pas encore été ajouté à la liste, donc c'est sous ces conditions qu'on l'ajoute. Avant d'ajouter un circuit, on fait appel à `calculer_longueur_opti` dessus pour lui associer sa longueur optimale. Il y a une exception. Si le sous ensemble possède un ou deux éléments, sa longueur optimale est respectivement la longueur d'un aller retour entre la base et ce sommet et la longueur du trajet en triangle reliant la base et les deux sommets (étant les seuls trajets possibles, il n'est pas nécessaire de les optimiser)

Si l'on tombe sur un circuit qui n'est pas réalisable, on ne fait pas d'appels récursif. En effet, on ne peut pas obtenir de circuit réalisable en ajoutant un ou plusieurs sommets à un circuit non réalisable.

Le déroulement de la fonction peut être modélisé par le schéma suivant :




```
void
calculer_longueur_opti(donnees_circuit * donnees_circ, donnees_map map)
```

Il s'agit de la fonction qui réalise le traveling salesman. Le choix a été fait de résoudre le problème de façon non-orientée, d'une part parce que cela permettait de réduire le nombre de contraintes et variables de décision en éliminant les doublons liés aux trajets i → j et j → i qui sont les mêmes dans le problème qui nous intéresse. On ne garde donc comme variables de décisions que $x_{i,j}$ pour i appartenant à l'ensemble des sommets et j ∈ i, qui vaut 1 si l'arc i → j est utilisé dans le chemin. Cela a tout de même posé le problème de l'indiçage des variables de décision, puisque l'on se retrouve avec un tableau des arcs empruntables en forme de triangle. Il a donc fallu user d'une fonction à deux paramètres (plus la taille de la matrice) pour effectuer la traduction entre un couple (i,j) et l'indice de l'arc reliant i et j.

Exemple d'indiçage des arcs pour un graphe à 6 sommets :

| arc de \ à | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | | 1 | 2 | 3 | 4 | 5 |
| 2 | | | 6 | 7 | 8 | 9 |
| 3 | | | | 10 | 11 | 12 |
| 4 | | | | | 13 | 14 |
| 5 | | | | | | 15 |
| 6 | | | | | | |

n° d'arc

Un problème de voyageur de commerce est normalement composé de trois jeux de contraintes : l'une assurant que chaque sommet a un prédécesseur, l'autre assurant que chaque sommet a une successeur, et la troisième assurant l'absence de sous-tours. Cette troisième contrainte peut être formulée de la façon suivante : chaque sous ensemble de l'ensemble des sommets (ni égal à l'ensemble lui-même ni vide) doit avoir au moins deux liens existant entre l'un de ses sommets et un des sommets appartenant à son complémentaire.

Notre hypothèse simplificatrice a permi de tout faire en un seul jeu de contraintes. Il correspond au troisième jeu de contraintes dans un voyageur de commerce orienté. Cela est suffisant. En effet on peut fusionner les deux premiers jeux de contraintes en "chaque sommet doit posséder au moins deux liens". Or, cela est géré par le troisième jeu de contraintes, puisque les singletons font partie des sous-ensembles.

La complexité de la mise en place des variables de décision et des contraintes est en $n\,2^n$ , n étant le nombre de sommets, puisqu'on a une contrainte portant potentiellement sur n éléments à appliquer par sous-ensemble de l'ensemble des sommets, et qu'un ensemble de cardinal n possède $2^n$ sous-ensembles.

## Résolution du partitionnement d'ensembles

Pour la résolution du problème de partitionnement d'ensembles nous avons procédé comme pendant les TD.

Les variables de décisions sont $x_i$ pour i entre 1 et le nombre de sous-ensembles parmi lesquels on a à choisir, valant 1 si le sous-ensemble i est "pris" dans le partitionnement.

Les contraintes sont telles que chaque sommet est "pris" par un et un seul sous-ensemble, c'est à dire que pour chaque sommet s la somme des variables de décisions correspondant aux sous-ensembles dans lesquels s'est inclut vaut 1.

# Analyse des résultats

Le programme tourne correctement sur les instances du dossier "A". On s'aperçoit que ces instances ont un ratio $\frac{capacitémoyenned'unesource}{capacitédudrone}$ relativement faible par rapport aux instances du dossier "B", c'est-à-dire que les tournées du drone sont assez courtes. Ainsi les appels à la fonction génératrice des tournées qui n'engendrent pas d'appels récursifs (car la tournée considérée est irréalisable) se font à une profondeur assez faible, et la génération n'a pas le temps d' "exploser". Les temps de calcul sont de l'ordre de la dizaine de seconde jusqu'à l'instance de taille 50.

En revanche, quand on s'attaque aux instances du dossier "B", on s'aperçoit que la récursivité n'était peut-être pas la meilleure idée.. en effet le nombre d'appels récursifs laissés en suspens est énorme et sur les grosses instances on en arrive à provoquer un arrêt du programme faute de mémoire.

En comparant avec ceux qui ont résolu le traveling salesman de façon brute, on n'a pas vraiment constaté d'avantage à utiliser glpk en ce qui concerne le temps de calcul. On peut conjecturer que sur de plus grosses instances on obtiendra un avantage, car la mise en place des contraintes est en complexité $n\,2^n$ ce qui à terme est nettement inférieur à $n!$ , et on peut raisonnablement supposer que glpk résout le problème avec une complexité meilleure que $n!$ . On comprend bien, avec toutes les opérations que l'on effectue avant la résolution du problème par glpk, qu'un test simpliste sur une quantité de l'ordre de la centaine sera plus efficace.

# Optimisations Possibles

## Calcul des sous circuits et de leur longueur associée

Nous aurions peut-être pu passer un pointeur sur problème glpk en paramètre à la fonction qui génère les tournées de façon à les parcourir sans occasionner un coût en mémoire énorme.

Nous aurions également pu éviter bon nombre d'appels récursifs en changeant notre façon de procéder lors de la génération. Plutôt qu'une structure ressemblant à un arbre binaire, nous aurions pu, à chaque appel, effectuer un appel récursif pour chaque circuit obtenu en ajoutant au circuit courant un sommet dont l'indice est compris entre l'indice de son dernier sommet et l'indice maximal. On aurait ainsi éliminé les appels "stériles" et on aurait probablement fortement gagné en temps de calcul et coût mémoire.

# Conclusions

Malgré un début relativement tortueux dû à l'explosion du coût en mémoire, le produit fini est satisfaisant en terme d'efficacité. Au final, nos difficultés se sont plus situées au niveau de l'implémentation que de la résolution crue du problème, notamment à cause d'une volonté têtue de vouloir résoudre le problème du voyageur de commerce à l'aide de glpk, avec un indiçage des variables peu intuitif.