UNIVERSITÉ DE NANTES
FACULTÉ DES SCIENCES
ET DES TECHNIQUES

Blade Flyer II : conquest of water

TP 4 Recherche Opérationnelle

Méthode de résolution exacte d'un problème de tournée de véhicule avec capacités et profits

---

Pagé Raphaël, Moustafa Ossama
Licence 3 Informatique
Année universitaire 2016/2017

Table des matières

Chapitre 1

Introduction

30 Mars, Année 3 du nouveau cycle. Clan Furiosa L'une des premières mesures d'Immortan Furiosa lors son accession à la tête de notre clan a été de normaliser, répertorier et rénover les points de pompage d'eau. Désormais chacun d'eux dispose d'un dispositif de collecte d'eau permettant de remettre au maximum leur niveau d'eau après chaque épisode pluvieux.
Ce projet a été couronné de succès. Mais, jaloux de notre prospérité, de nombreuses bandes armées ce sont organisées afin de prendre d'assaut nos points d'eau. Nous avons donc mis en place des défenses des plus efficaces autour de chacun d'eux, les rendant virtuellement imprenable. En réponse à cela, les bandes s'en prennent désormais à nos convois chargés de transporter l'eau.
Furiosa, en réponse à cela, a commandé en grand secret la préparation d'un drone muni d'un réservoir qui servira de nouveau moyen de transport d'eau, prenant à coup sur les bandits par surprise. Mais le temps presse, les réserves d'eau s'amenuisant dangereusement de jour en jour. L'équipe de planification nous donne un mois pour mettre en place le drone.
Ce rapport vous présentera l'algorithme permettant de calculer la trajet que le drone devra suivre pour optimiser le temps de parcours nécessaire afin de visiter tout les points de pompage.
Merci de bien vouloir lire le fichier *README* qui vous informera du contenu du dossier qui vous est présenté.

Chapitre 2

Description des structures de données choisies pour l'implémentation

2.1 Méthode utilisée

Bien entendu, notre drone a un réservoir d'une capacité limitée. Visiter tout les points d'eau en une seule fois est impossible. Il nous faut donc trouver la suite de tournées (une tournée correspond à un trajet du drone, c'est à dire l'ordre de visite des points de pompage avant de revenir à la base) offrant la plus petite distance à parcourir.

La méthode que nous avons adopté consiste donc à énumérer dans un premier temps les différents regroupements possibles des points de pompage puis, pour chacun de ces regroupements, déterminer la meilleure permutation des indices des points de pompage (c'est à dire la permutation offrant la plus courte distance à parcourir).
Il ne nous reste ensuite plus qu'à faire une sélection parmi les regroupements, de telle façon que tout les points de pompage soit visités, et la distance parcourue soit minimale.

2.2 Structures de données

Vous pouvez trouver une documentation exhaustive des différentes structures de données utilisées dans le fichier *Documentation_ bladeFlyer.pdf* (généré grâce à *Doxygen* [1]), mais pour résumer nous avons décidé de travailler avec deux classes et une structure.

La première classe, Tournee, représente une tournée que le drone pourrait faire.
Elle a pour attributs deux vecteurs d'entiers et un entier. Le premier vecteur contient les indices des points d'eau visités et le deuxième la permutation des ces indices offrant la plus petite distance à couvrir. L'entier correspond à la plus petite distance à parcourir pour visiter les points d'eau (c'est à dire la distance totale du trajet suggéré par la permutation optimale).
Cette classe contient un ensemble de méthodes permettant d'accéder et de mettre à jour la valeur des ces attributs, ainsi qu'une méthode permettant de déterminer la permutation offrant la plus petite distance à parcourir, et une méthode permettant de calculer la distance parcourue pendant une tournée.

La deuxième classe, Enumeration, représente une énumération des tournées réalisables.
Elle a pour attributs un vecteur de Tournee, ainsi qu'une Donnee (structure décrite un peu plus loin) contenant les données du problème.
Cette classe contient un ensemble de méthodes permettant d'accéder et de mettre à jour la valeur des ces attributs, ainsi qu'une méthode permettant de déterminer si la capacité restante du réservoir du drone lui permet de visiter un point de pompage, et d'une méthode permettant de déterminer l'ensemble des différentes tournées que le drône peut réaliser sans devoir retourner à la base pour vider son réservoir.

Nous disposons également d'une structure, Donnee, permettant de contenir les données du problème. Elle contient quatre entiers, représentant le nombre de lieux ayant une incidence sur le problème (c'est à dire la base et les points de pompage), la capacité du réservoir du drone, la quantité d'eau disponible à chaque point de pompage ainsi que le distancier, répertoriant le distance entre chaque lieu.

La raison pour laquelle nous avons décidé de travailler en c++ est que nous souhaitions utiliser les avantages de la programmation objet (et que nous avions déjà quelques bases dans ce domaine). Nous avons affaire dans ce projet à plusieurs entités (notre drone, les points de pompages, les tournées, le regroupements de ces tournées...), la programmation objet nous est donc venu assez naturellement à l'esprit.
Les futurs changements et améliorations de notre projet (par exemple l'implémentation d'armes sur le drone, ou d'une autonomie limitée du drone) pourront ainsi être ajoutées assez facilement au code existant.
Il est tout de fois vrai qu'après coup nous nous sommes rendu compte que certaines classes auraient pu être implémentées sous forme de structures, et que nous nous sommes peut-être laissé emporter par notre enthousiasme de la programmation objet (en particulier la classe Enumeration, qui aurait pu être simplement une structure comportant un vecteur de Tournée, utilisé par un ensemble de fonction).
Il serait intéressant de discuter des avantages et inconvénients de la programmation objet dans notre cas.

Chapitre 3

Description de l'algorithme d'énumération des regroupements possibles des points de pompage

3.1 Générer un powerset

L'algorithme permettant de générer un vecteur contenant les différentes tournées que le drone peut effectuer peut être trouvé dans le fichier *Enumeration.cpp*. Il s'agit de la méthode *powerset* de la classe *Enumeration*. Une copie du code peut-être retrouvé dans la figure 3.1.
Nous avons la chance que les points de pompages soient numérotés avec des entiers, sans discontinuités et en partant de 1. Il nous suffit donc de mettre en place un algorithme permettant de générer l'ensemble des parties d'un ensemble (*powerset* en anglais) représentant une suite d'entier de taille finie et partant de 1. Fort heureusement c'est un problème courant et il existe de nombreuses documentations à ce sujet (par exemple [3]).
Nous avons opté pour une approche itérative du problème puisqu'il nous paraissait ainsi plus évident d'y ajouter les différentes vérifications nécessaire avant d'ajouter un point d'eau à la tournée (en particulier vérifier si le réservoir du drone lui permet de visiter un point de pompage donné).
Pour mettre en place cet algorithme, nous sommes partis de l'idée d'utiliser une pile qui grandit ou rétrécit selon nos besoin. L'algorithme va fonctionner ainsi :
La pile va grandir jusqu'à ce que le dernier élément de l'ensemble soit atteint.
Lorsque c'est le cas, l'avant dernier élément de la pile va prendre la valeur de l'entier lui succédant et la pile va rétrécir.
Il faudra alors répéter ces opérations jusqu'à ce que la pile ne puisse plus rétrécir, et créer une tournee avec le contenu de la pile après chacune des étapes.
Un exemple sera sans doute plus parlant. Prenons l'ensemble {1, 2}.
Au début, la pile n'aura que {1} . 1 n'est pas le dernier élément de l'ensemble, la pile peut donc grandir : {1,2}
2 est le dernier élément de l'ensemble. 1 prend donc la valeur de l'entier qui lui succède, et la pile rétrécit : {2}
2 est la dernière valeur de l'ensemble, mais la pile ne peut plus rétrécir. L'algorithme a donc généré le powerset de l'ensemble 1,2.

```
/*! \brief permet de determiner l'ensemble des différentes tournées que le drône peut réaliser sans vider son
réservoir
    \param n taille de l'ensemble à traiter
*/
void Enumeration::powerset (int n){
    int stack[n+1];
    int position;

    stack[0]=0; //0 represente la base et 0 ne fait pas parti du set
    position = 0;

    bool done = false;
    while(!done) //on va utiliser un tableau qui va aggrandir et retrecir le set
    {

            if (stack[position]<n) //si l'on a pas dépassé la borne du set
            {
                stack[position+1] = stack[position] + 1; //on fait grandir le tableau
                position++;
            }

            else //on revient en arrière et on recommence
            {
                stack[position-1]++;
                position--;
            }

            if (position==0) // quand position arrive à 0, on a exploré toute les options
                done = true;

            if (capaciteSuffisante(stack,position)) //si le reservoir du drone permet de visiter le point d'eau
                addToEnum(stack,position); //alors on ajoute la tournée à l'énumeration de l'ensemble des
tournées possible

    }
    return;
}
```

FIGURE 3.1 – Algorithme permet de générer le powerset d'en ensemble d'une taille donnée

3.2 Déterminer la meilleure permutation d'une tournée

Le problème c'est que cet algorithme génère le powerset selon un ordre lexicographique. Or il est probable, voir même évident, que l'ordre dans lequel les indices des points d'eau sont alors présentés n'offre pas un trajet permettant au drone de parcourir une distance minimale.
Il nous faut donc déterminer, pour chacune des tournées que nous allons ajouter à l'Enumeration, la permutation offrant une distance optimale.

Après quelques minutes passées à considérer ce problème, nous avons enfin compris que celui-ci n'était autre que l'un des plus célèbres du monde informatique...le problème du voyageur de commerce ! En effet, nous souhaitons visiter exactement une fois tout les points de pompage présents dans la tournée, dans l'ordre offrant la plus petite distance.
La première solution qui nous est venue à l'esprit (et que nous avons implémenté dans le projet), est le brute force. Il suffit d'examiner toute les permutations possible, de calculer la distance nécessaire pour parcourir chacune d'entre elles, et de garder la permutation offrant la plus courte tournée (vous pouvez retrouver notre version de l'algorithme dans le fichier *Tournee.cpp*. Il s'agit de la méthode *calculePlusPetiteDistancePerm* de la classe Tournee. Il est également décrit par la figure 3.2).

Il nous était tout de fois évident que comme toute approche utilisant le brute forcing, notre algorithme ne serait plus du tout efficace si le nombre de point de pompage à visiter augmentait. C'est à dire si nous avions accès à un nombre important de point de pompage (ce qui, dans notre situation post-apocalyptique, nous semble pour le moins incongrus mais souhaitable), si la quantité d'eau disponible à chaque point de pompage diminuait ou si la capacité du réservoir de notre drone augmentait (ces deux derniers cas autorisant notre drone à visiter plus de points d'eau en une tournée).

Une recherche exhaustive comme la notre va avoir une complexité en O(n !). C'est à dire que si le calcul d'un chemin prend une microseconde, alors le calcul de tous les chemins pour 10 points est de 181 440 microsecondes soit 0,18 seconde mais pour 15 points, cela représente déjà 43 589 145 600 microsecondes soit un peu plus de 12 heures.

```
/*! \brief calcule la permutation offrant la plus petite distance à parcourir en brute force
        \param tour indices des points d'eau visités
        \param d données du problème
        \return meilleur_tour permutation minimale
*/
vector<int> Tournee::calculePlusPetiteDistancePerm( vector<int> tour, Donnee d)
{
        int meilleur_distance = calculeDistanceTour(tour, d);
        vector<int> meilleur_tour = tour;

        while(next_permutation(tour.begin(), tour.end())) //on va simplement examiner toute les permutations
possible, et mettre à jour la valeur de la distance  minimale
                {
                    int distance = calculeDistanceTour(tour, d);
                    if ( distance < meilleur_distance )
                        {
                                meilleur_distance=distance;
                                meilleur_tour=tour;
                        }

            }
          return meilleur_tour;
}
```

FIGURE 3.2 – Algorithme permet de résoudre une instance du problème du voyageur de commerce

Il existe bien sur des manières plus efficaces de trouver une solution exacte au problème du voyageur de commerce (en utilisant la programmation dynamique Held et Karp ont par exemple démontré que la complexité du problème était en $O(n^2 2^n)$ [2]). La programmation linéaire est aujourd'hui la manière la plus efficace de résoudre ce problème de manière exacte, et c'est une approche que nous aurions aimé aborder, si nous avions eu un peu plus de temps.

Mais le problème du voyageur de commerce est un problème dont on ne connaît de toute façon pas d'algorithme en temps polynomial ([4]). Toutes les solutions que nous pourrions de toute facon trouver seront de toute manière inefficaces à un moment ou à un autre.
Nous recommandons donc de ne pas compter sur un seul drone pour visiter tout les points de pompages, mais plutôt de réussir à s'en procurer un autre et de distribuer les tournées entre les différents drones.

Chapitre 4

Description de l'algorithme d'énumération des tournées

4.1 Résolution du problème avec GLPK

Une fois la liste de toute les tournées possibles déterminée et triée, il ne nous reste plus qu'à trouver quelles tournées choisir afin de visiter tout les points de pompage tout en gardant une distance parcourue minimale.

Nous avons décidé de modéliser ce problème en programmation linéaire. Il nous suffit d'associer une variable binaire à chacune des tournées, qui vont représenter la décision de choisir cette tournée ou non. La fonction objectif s'écrira alors :

$\sum_{i=1}^{nbVar}(l_i * x_i)$

avec *NbVar* représentant le nombre de tournées possible, et l_i la plus petite distance requise afin de parcourir tout les points de la tournée x_i.
Il nous faut ensuite vérifier que les points d'eau ne soient visités qu'une seule fois. Pour cela, il suffit d'additionner chacun des x_i où un point d'eau j serait présent et les mettre à l'égalité avec 1.

Vous pourrez retrouver notre version de l'algorithme dans le fichier *partitionEnsemble.cpp* sous la forme de la fonction *partitionEnsemble*.
Elle prend en paramètre une Enumeration et un entier représentant le nombre de lieux du problème à traiter.
Le nombre de variable du problème va être en réalité le nombre de tournées possible, et le nombre de contraintes le nombres de lieux minus la base.
Il s'agit ensuite d'un programme assez similaire à celui que nous avons eu l'occasion de mettre en place lors du *TP 3*, et il est en grande partie basé sur son modèle. La seule partie de l'algorithme un tant soit peu "original" est la déclaration des contraintes. Il s'agit en fait de créer trois vecteurs d'entiers contenant respectivement les lignes, colonnes et valeurs de la matrice creuse, puis de parcourir l'ensemble des tournées. Pour chacune d'entre elles, il suffit d'ajouter les indices des points de pompage présent dans la tournée au vecteur des lignes de la matrice (puisque ces indices vont représenter les numéros des contraintes), d'ajouter l'indice de la tournée que nous sommes en train d'examiner au vecteur des colonnes de la matrice (puisque cet indice va représenter la numéro de la variable) et de mettre sa valeur à 1 dans le vecteur des valeurs de la matrice.

Chapitre 5

Analyse des résultats

Vous pourrez retrouver les résultats de l'exécution de notre programme avec différents fichier de données dans le fichier *resultats.txt* (les tests ont été réalisés en utilisant les ordinateurs de l'université).

Les fichiers de données sont répartis dans deux dossiers séparés, *A* et *B*. Dans chacun des cas, nous avons considéré notre algorithme comme n'étant plus efficace s'il mettait plus d'une minute à s'executer.

Dans le dossier A, la seule différence entre deux fichiers de données est l'augmentation du nombre de points de pompage. Le premier fichier, *VRPA10.dat* présente une situation avec dix points de pompage, tandis que le dernier fichier, *VRPA50.dat* présente une situation à cinquante points de pompage.
Notre algorithme a été capable de résoudre un problème efficacement jusqu'au *VRPA30.dat*, fichier présentant une situation avec trente points de pompage. Dans cette situation, l'algorithme permettant de générer le powerset optimisé a mis 52.591531 secondes a s'exécuter, GLPK a mis 5.831372 secondes a résoudre le problème et le code a mis 58.422904 secondes au total à s'exécuter.
On peut donc facilement comprendre ici que c'est la méthode utilisant le brute force pour générer le powerset qui limite notre algorithme. Il aurait été intéressant d'essayer de résoudre le problème avec GLPK pour voir si la limite d'efficacité de notre algorithme aurait été plus haute.

Le dossier B nous présente des fichiers de données où la capacité du réservoir du drone a été augmenté de trois litres par rapport aux fichiers du dossier A, et où les réserves d'eau des points de pompages sont bien inférieure.
Notre algorithme est devenu inefficace bien plus tôt que précédemment. En effet, dès le fichier *VRPB20.dat*, le code a mis 422.282866 secondes au total à s'exécuter. Mais cette fois-ci le facteur limitant notre programme n'était plus l'algorithme du powerset...mais la résolution avec GLPK !
L'algorithme du powerset a mis 55.226971 secondes a s'exécuter tandis que GLPK a mis 367.055893 secondes a résoudre le problème.
Et c'est normal. Dans le cas où le nombre de points d'eau maximum à visiter lors d'une tournée augmente, le problème de partitionnement d'ensemble devient de plus en plus complexe...même pour la programmation linéaire.

Chapitre 6

Conclusion

Même si l'on a vu que notre algorithme se retrouve assez vite limité dés que les données deviennent trop importante, on peut le considérer comme étant tout de même assez satisfaisant dans notre situation. En effet, le jour où nous aurons trente points de pompage à notre disposition, j'espère sincèrement que nous aurons plus d'un drone à notre disposition.

D'autant que la raison pour laquelle le problème du voyageur de commerce et le problème de partitionnement d'un ensemble limitent notre programme est que ces deux problèmes sont NP-Complet. Il ne sera donc possible de les optimiser qu'à un certain degré.

Toutefois il y aurait plusieurs pistes à explorer si l'on souhaitait optimiser ce programme, comme par exemple explorer la piste de la programmation linéaire ou des heuristiques pour résoudre le problème du voyageur de commerce.

Bibliographie

[1] *Doxygen*. URL : `http://www.stack.nl/~dimitri/doxygen/index.html`.

[2] R. M. Karp M. Held. *A Dynamic Programming Approach to Sequencing Problems*. Journal of the Society for Industrial et Applied Mathematics, 1962.

[3] *Pro Programming, Powerset Algorithm in c++*. URL : `http://proprogramming.org/powerset-algorithm-in-c/`.

[4] *TSP, Traveling Salesman Problem*. URL : `http://www.math.uwaterloo.ca/tsp/`.