2019

HVRP

TP3 – OAD

JAEL VAVASSEUR – CLÉMENT MESNIL

ISIMA |1 rue de la Chebarde, 63170 Aubière

Compte rendu

TP 3

Outils d’aide à la décision

Jaël Vavasseur – Clément Mesnil Philippe Lacomme

Table des matières

Introduction 3
1. Descriptif des points nécessaires à l'optimisation d'un problème HVRP 4
Les structures utilisées 4
Construction de solutions initiales 5
Implémentation de la fonction SPLIT 6
Conception d'une recherche locale efficace 7
2. Description algorithmique des fonctions d'optimisation 8
La procédure operateur_2_opt() 8
La procédure operateur_2_opt_inter_arrivee() 9
La procédure deplacement_sommet() 10
La procédure SPLIT() 11
3. Etude des résultats 12
Conclusion 13

Introduction

Le but de ce TP est d'implanter un outil permettant de s'initier aux problèmes HVRP. Ce sont des problèmes de transport, comme le plus connu d'entre eux : le voyageur de commerce ; mais dans notre cas, une notion de capacité du véhicule est à prendre en compte. Des quantités seront à récolter OU à livrer. De plus, contrairement aux problèmes de VRP, les camions auront ici des capacités et des coûts variables et fixes différents selon le type du camion.

1. Descriptif des points nécessaires à l'optimisation d'un problème HVRP

Avant toutes choses, nous avons dû définir les structures que nous avons utilisées afin de stocker notre problème, notre solution, et diverses autres structures intermédiaires pour faciliter la résolution du problème :

Les structures utilisées

Notre structure **T_instance** contient différents éléments qui nous permettront de traiter des instances. Ces instances sont sous forme de fichiers textes bruts.

- nb_clients : le nombre de clients à desservir
- qte[] : tableau des quantités à prélever
- distance[][] : matrice des distances
- nbtypecam : nombre de types de camions de l'instance
- V_som[6] : tableau permettant de mémoriser les plus proches voisins d'un sommet
- Liste_types[] : tableau contenant les types de camions

```
typedef struct T_instance {
    int    nb_client = 0;
    int    qte[nmax + 1] = { 0 };
    double distance[nmax + 1][nmax + 1];
    int    nbtypecam = 0;
    int    V_som[6] = { 0 };
    T_camion liste_types[nmaxtype];
}T_instance;
```

La structure **T_camion** contient des informations relatives aux camions que l'on utilisera.

```
typedef struct T_camion {
    int   nb = 0;                    // nombre de camions
    int   capacite = 0;              // capacite de ce type de camions
    int   cf = 0;                    // coût fixe ex: 10€
    float cv = 0;                    // coût variable ex: 0.1€/km

}T_camion;
```

La structure **T_solution** contient une solution associée à une instance :

- cout_total : coût final de la solution en euros
- nb_tournees : nombre de tournées
- Liste_tournees[] : liste des tournées à réaliser

```
typedef struct T_solution {
    int       cout_total = 0;
    int       nb_tournees = 0;
    T_tournee liste_tournees[nmaxtournee];
}T_solution;
```

Enfin, une solution initiale appelée tour géant est réalisée, puis est décomposée en **T_tournee** :

- type_camion : indice permettant de connaître le type du camion sur la tournée
- nb_sauts : nombre de villes à visiter pendant la tournée
- liste_sauts[] : matrices des sauts effectués dans la tournée
- cout : coût de la tournée en euros
- volume : volume déplacé durant la tournée

```
typedef struct T_tournee {
    int   type_camion = 0;
    int   nb_sauts = 0;
    int   liste_sauts[100] = { 0 };

    double cout = 0.;
    int   volume = 0;
}T_tournee;
```

Construction de solutions initiales

Afin de construire des solutions initiales et donc des tours géants, nous avons implémenté 3 heuristiques différentes.

Nous avons premièrement implémenté une méthode qui, en partant d'un sommet, trouve ses 5 voisins les plus proches en termes de distance. Ensuite, une méthode permettant de créer un tour géant en choisissant le sommet suivant comme étant le plus proche du sommet en question.

Ensuite, une deuxième heuristique a été mise en place, similaire à la première à la seule différence que le sommet suivant choisi a de très grandes chances d'être le plus proche. Toutefois, il reste probable de ne pas choisir pour le sommet suivant, le sommet le plus proche du sommet en question.

Enfin , nous avons implémenté l'heuristique la plus simple qu'il était possible de faire : le tour géant est construit par ordre croissant des numéros des sommets. On ne prend donc pas en compte la distance entre les différents sommets.

Implémentation de la fonction SPLIT

Toute solution du voyageur de commerce peut se découper optimalement en solution du VRP en $O(n)$. C'est le but du SPLIT. A partir d'un tour géant, on met sur les arcs possibles le coût de la tournée avant d'appliquer sur le graphe un algorithme de plus court chemin. Le découpage obtenu sera forcément optimal.

Afin de découper un tour géant en tournées, nous avons défini une nouvelle structure de données, un label, qui va garder en mémoire tous les tests de route à prendre, selon le type de camion sélectionné. Ces labels seront très utiles pour remonter le graphe créé et ainsi trouver le chemin critique. On y sauvegarde aussi la liste des précédents sauts ainsi que la liste des camions utilisés pour ces sauts.

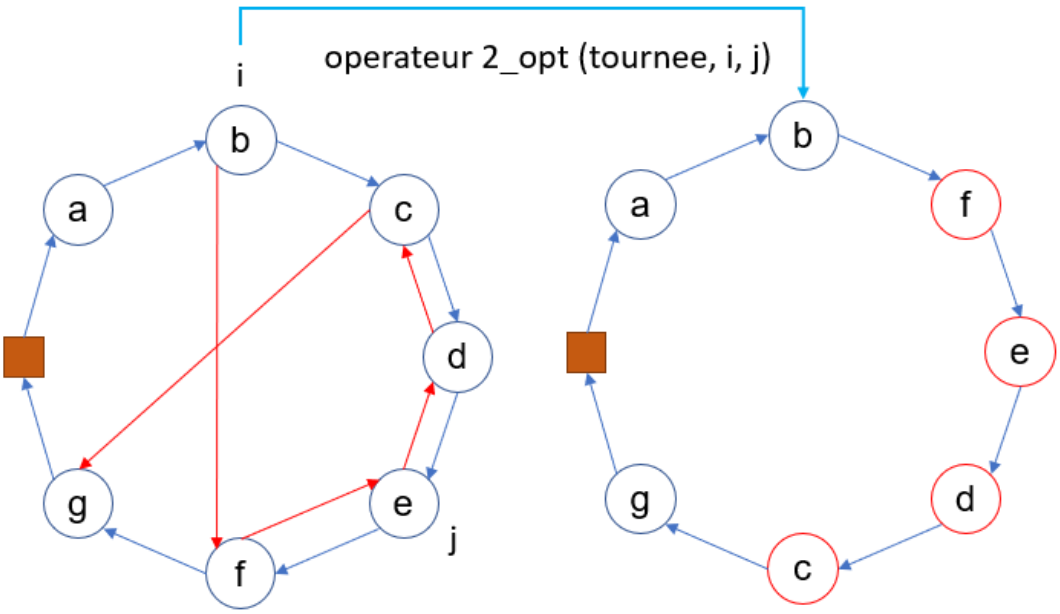
```
typedef struct T_label {
    double prix = 0;
    int capacite = 0;
    int pere[100] = { 0 };
    int nb_peres = 0;
    int reste_camions[10] = { 0 };
    int nb_sauts[100] = { 0 };
    bool labels =  false;
    int type_camion[100] = { -1 };

}T_label;
```

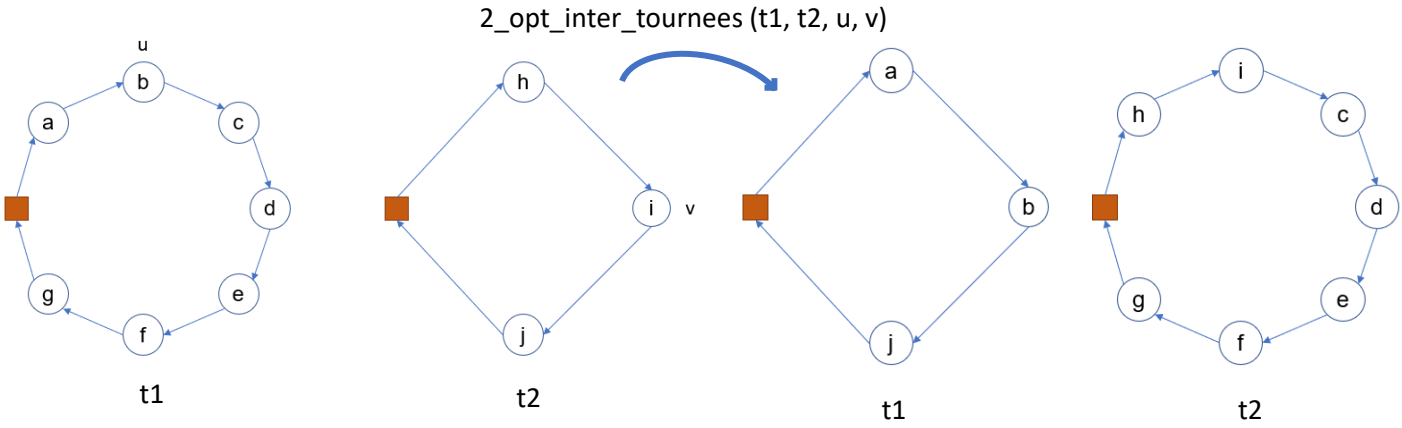
Les labels sont stockés dans une liste correspondant à chaque saut et ils y sont comparés à ceux existants pour ce nœud. A la fin, nous choisissons un label dans le dernier nœud qui nous permettre de remonter le chemin critique. Ce label est choisi arbitrairement puisque s'il existe plusieurs labels au dernier saut c'est qu'ils ne se dominent pas 2 à 2. Or il n'existe pas de relation d'ordre total sur ces labels (nous n'en avons pas créé ici puisque cela ne semblait pas pertinent).

Conception d'une recherche locale efficace

La recherche locale nous permet d'améliorer nos solutions. Pour ce faire, on utilise un grand nombre de fois (en respectant un nombre d'itérations maximal) plusieurs opérateurs permettant de trouver de potentielles solutions améliorées. Nous avons utilisé un opérateur **2_opt** qui, à partir d'une tournée, d'un sommet i et d'un sommet j, transforme la tournée en fixant saut suivant le sommet i au sommet j directement, puis en inversant les sauts intermédiaires.



Un deuxième opérateur, assez proche du premier a également été utilisé : c'est l'opérateur **2_opt_inter_tournees**. Le principe est le même que l'opérateur précédent, mais la modification des sauts se fait au niveau de deux tournées simultanément. En l'utilisant, on obtient ainsi 2 tournées filles ayant toutes deux un morceau des tournées mères.



La fonction recherche_locale va lancer sur notre solution une série d'opérateurs, en l'occurrence ici 2-opt, 2-opt inter tournée et déplacement. Ces opérateurs ont une probabilité de se lancer qui augmente plus l'opérateur est efficace et inversement s'il n'améliore pas la solution.

2. Description algorithmique des fonctions d'optimisation

Nous allons dans cette partie décrire plus précisément le fonctionnement des fonctions que l'on a décrites ci-dessus.

La procédure operateur_2_opt()

```
operateur_2_opt (instance, tournee)
{

        initialiser les variables des coûts et de l'itération;

                POUR chaque sommet allant de 1 à (nbsauts_tournee) -2 FAIRE {

                        stocker la distance entre le sommet i et i+1;
                        POUR chaque sommet allant de i+2 à (nbsauts_tournee) FAIRE {

                                stocker les distances des axes qui seront supprimés et ajoutés;
                                calculer la variation de distance;
                                SI la nouvelle distance meilleure ALORS {

                                        modifier la distance totale de la tournée;
                                        effectuer la rotation des sommets sur la tournée;
                                }
                        }
                }
}
```

La procédure operateur_2_opt_inter_arrivee()

```
operateur_2_opt_INTER_TOURNEE (instance, tournee_1, tournee_2)
{
        initialiser les variables temporaires utilisées pour vérifier l'amélioration des solutions;
        POUR chaque sommet allant de 1 à (nbsauts_tournee_1) -1  FAIRE {
                POUR chaque sommet allant de 0 à i FAIRE {
                        stocker distance par rapport au sommet prochain;
                        stocker la quantité à ramasser au sommet;
                }
                POUR chaque sommet allant de 1 à (nbsauts_tournee_1) -1  FAIRE {
                        POUR chaque sommet allant de 0 à i FAIRE {
                                stocker distance par rapport au sommet prochain;
                                stocker la quantité à ramasser au sommet;
                        }
                        ajouter le sommet j+1 de tournee_2 à tournee_1 (dist et qté);
                        POUR chaque sommet supprimé de la tournee_2 FAIRE {
                        stocker distance par rapport au sommet prochain;
                        stocker la quantité à ramasser au sommet;
                        }
                         ajouter le sommet i+1 de tournee_1 à tournee_2 (dist et qté);
                        POUR chaque sommet supprimé de la tournee_1 FAIRE {
                                stocker distance par rapport au sommet prochain;
                                stocker la quantité à ramasser au sommet;
                        }
                        calculer les coûts des potentielles tournées;
                        SI les tournées sont mieux et que les capas sont respectées ALORS {
                                actualiser les coûts des tournées;
                                faire la rotation des sommets à visiter ;
                        }
                }
        }
}
```

La procédure deplacement_sommet()

```
deplacement_sommet(instance, tournee) {
   initialiser les variables utilisées;
   POUR chaque sommet de la tournée FAIRE {

         POUR chaque saut dans la tournée FAIRE {

               SI les indices sont différents et ne se succèdent pas ALORS {

                     calculer la distance de la tournée en déplaçant les sommets;
                     SI la nouvelle distance totale est plus faible que l'ancienne ALORS {

                           Mettre à jour la distance
                           Sauvegarder les indices
                     }
               }
         }
   }
   SI l'on a amélioré la tournée ALORS {

         décaler les sommets d'un cran;
         mettre à jour le coût de la tournée;
    }
}
```

La procédure SPLIT()

```
SPLIT(Instance, tour géant)
        Pour tous les sommets dans le tour géant faire
                Pour tous les sommets dans le tour géant faire
                        Pour tous les types de camions de l'instance
                                Pour tous les labels dans ce nœud
                                        Calculer tous les labels possibles pour les nœuds suivants
                                        Trier les labels et vérifier la domination entre labels
                                Fait ;
                        Fait ;
                Fait ;
        Fait ;
        // on a créé le graphe avec tous les labels
        // il faut ensuite trouver le chemin critique
        //on le retrouve facilement dans la structure du label
        Remonter le chemin critique en prenant le meilleur label du dernier saut
Fin ;
```

3. Etude des résultats

La fonction SPLIT nous donne régulièrement plusieurs solutions valides, nous aurions pu nous en servir pour créer des populations de solutions et ainsi améliorer la population de solution avec un algorithme génétique, mais nous avons manqué de temps pour réaliser cette partie.

Nous avons donc choisi de prendre la première solution valide arbitrairement et d'essayer de l'améliorer avec nos 3 opérateurs.

Voici un petit tableau de résultats de notre solution sur quelques instances :

instance	nb ville	coût total solution
HVRP_DLP_75 (Paris)	20	1444
HVRP_DLP_55	56	6404
HVRP_DLP_92	34	8281
HVRP_DLP_93	39	487
HVRP_DLP_94	46	1765

On trouve des résultats assez bons, ce qui montre que notre code comporte surement des erreurs… De plus, nous avons fait tourner notre code sur des plus grosses instances mais le temps d'attente était interminable du fait de la fonction dominer, qui est très importante et détermine si les labels sont supprimés ou non. Il aurait fallu en trouver une plus restrictive afin que le nombre de label soit plus petit et le temps de calcul en serait sorti drastiquement réduit.

Conclusion

Le problème HVRP est un problème très complexe mais qui devient de plus en plus d'actualité, il apparait notamment indispensable pour des firmes telles qu'Amazon pour optimiser ses livraisons et aussi pour La Poste, qui cherchent toujours à optimiser jusqu'au moindre petit kilomètre.

Ce TP nous a montré la complexité de ce genre de problème et surtout la vitesse à laquelle la complexité peut augmenter pour des problèmes qui n'ont pas l'air si complexes à première vue. Nous avons rencontré plusieurs difficultés durant ce TP et notamment la fonction SPLIT nous a pris beaucoup de temps, elle pourrait être optimisée en temps grâce notamment à la mise en place d'une fonction domine qui soit plus restrictive, tout en n'étant pas trop restrictive non plus, ce qui aurait pour conséquence de ne trouver aucune solution. Nous ne gérons de plus aucune exception, par exemple si une instance demande de prélever plus de quantité que la capacité des camions.

Nous pourrions aussi améliorer ce SPLIT et les solutions en permettant aux camions de prélever une partie seulement de la charge sur un nœud, cela permettrait notamment d'utiliser moins de ressources (ici des camions).