Outils d'aide à la décision TP3
HVRP

Arquillière Mathieu - Zangla Jérémy

Table des matières

Table des figures

Liste des Algorithmes

1 Introduction

Notre problème ici est un problème de tournées de véhicules. Il faut trouver des algoritmes nous rapprochant de solutions qui déterminent les tournées que doivent faire des véhicules afin de livrer ou récupérer des marchandises chez des clients.

2 Heuristiques de construction de solution

Pour générer des solutions grâce à la fonction SPLIT qu'on détaillera plus loin, on a besoin *d'heuristiques de construction de solution initiales*. Celle-ci permettent de générer un *tour géant* à partir d'une instance d'un problème. Ce *tour géant* est un tableau contenant les sommets, les clients de l'instance, et selon l'ordre de ce tableau, la fonction SPLIT générera une solution plus ou moins efficace. On implémentera et testera 3 méthodes pour générer ce tour géant.

2.1 Tour géant aléatoire

2.1.1 Principe

Cette méthode consiste simplement à générer un tour géant aléatoirement. Il faut donc créer un tableau d'une taille n (nombre de sommets) contenant tous les nombres de 1 à n répartis aléatoirement. Pour optimiser, on créé un tableau de la forme $\forall i \in 1,...,n$ on a $T[i] = i$. On génére un indice *ind* aléatoire et on insère dans le tour géant $T[ind]$, puis on échange cet élément dans T avec le dernier élément et on réduit la taille du tableau de 1.

2.1.2 Implémentation

FIGURE 1 – Fonction de génération de tour géant - méthode aléatoire

```
/*
  Fonction qui permet de generer un tour geant de manière totalment aléatoire
  entrees : taille_tour, taille de la tournee à générer
  sorties : tournee, tableau d'entier représentant les sommets
*/
void generer_tour_geant_alea(int taille_tour, int * tournee)
{
  // On se sert d'un tableau dont on réduit la taille et on inverse le dernier element avec l'element qu'on a choisi
    aléatoirement
  int max = taille_tour;
  int tab_alea[nb_max_sommets];
  for (int i = 0; i < taille_tour; i++)
    tab_alea[i] = i;

  for (int i = 0; i < taille_tour; i++)
  {
    int ind = rand() % max;
    tournee[i] = tab_alea[ind];

    tab_alea[ind] = tab_alea[max - 1];
    max--;
  }
}
```

2.2 Plus proche voisins

2.2.1 Principe

Cette méthode pour générer le tour géant consiste à partir du sommet initial et de rajouter au tour géant son voisin le plus proche, ensuite on rajoute le voisin le plus proche à ce voisin etc. Cette méthode s'est révélée beaucoup plus efficace que la méthode précédente.

2.2.2 Implémentation

FIGURE 2 – Fonction de génération de tour géant - méthode aléatoire

```
/*
  Fonction qui génère un tour géant avec la méthode des plus proches voisins
  entrees : instance, instance dont on veut générer une solution
  sorties : T, tour géant
*/
void generer_tour_geant_ppv(t_instance& instance, int * T)
{
  T[0] = 1; // On initialise l'algo pour démarrer au premier sommet
  int L[nb_max_sommets + 1]; // Liste contenant les sommets
  for (int i = 0; i < instance.nb_sommets + 1; i++)
    L[i] = i + 1; // Chaque sommet est représenté par un nombre
  bool M[nb_max_sommets + 1]{0};
  int pv0 = 0;
  int x, y;
  int Px = 0; // Index du premier sommet dans la liste L
  int nr = instance.nb_sommets - 1; // Nombre de sommets pas encore traites


  for (int i = 1; i < instance.nb_sommets; i++)
  {
    x = T[i - 1]; // Sommet de départ
    // Suppression du sommet dans la liste
    L[Px] = L[nr]; // On retire l'ancien sommet de la liste.

    int pv0 = 0;
    int v0 = L[0];
    for (int j = 1; j < nr; j++) // Pour chaque sommet restant (rappel : on a retiré x)
    {
      y = L[j]; // Sommet actuel
      if (!M[y] && instance.distances[x][y] < instance.distances[x][v0]) // Si la distance est plus
    courte, on insere le sommet
      {
        v0 = y; // insertion de l'élément choisi
        pv0 = j;
      }

    }

    // Boucle de choix du voisin
    T[i] = v0;
    Px = pv0;
    M[v0] = true;

    nr--;
  }
}
```

2.3 Plus proche voisins randomisés

2.3.1 Principe

Cette méthode consiste à prendre les 5 plus proches voisins du sommet initial et de choisir aléatoirement dans ces sommets lequel on ajoute dans le tour géant. Cependant l'aléatoire n'est pas uniforme. En effet, on trie ces 5 sommets (en fonction de la distance avec le sommet initial) et le premier de cette liste à 80% de chance d'être séléctionné. Puis, le deuxième, lui, à 80% de chance des 20% restants d'être séléctionné, etc. Une fois le sommet choisi, on ré-exécute le même algorithme sur lui et on itère de cette façon jusqu'à avoir tous les sommets dans le tour géant.

2.3.2 Implémentation

FIGURE 3 – PPVR - 5 plus proches voisins

```
/*
  Fonction qui génère un tour géant avec la méthode des plus proches voisins randomisés. On choisit les 5 plus proches voisins, puis on ajoute
  dans le tour géant un de ces voisin aléatoirement, avec une probabilité de 80% pour le premier, 80% des 20% restants pour le deuxième etc
  entrees : instance, instance dont on veut générer une solution
  sorties : T, tour géant
*/
void generer_tour_geant_ppvr(t_instance& instance, int * T)
{
  T[0] = 1; // On initialise l'algo pour démarrer au premier sommet
  int L[nb_max_sommets + 1]{}; // Liste contenant les sommets
  for (int i = 0; i < instance.nb_sommets + 1; i++)
    L[i] = i + 1; // Chaque sommet est représenté par un nombre
  int V[5]; // Liste des 5 plus proches voisins
  int pV[5]; // Position dans L des 5 plus proches voisins

  int x, y;
  int Px = 0; // Index du premier sommet dans la liste L
  int nr = instance.nb_sommets - 1; // Nombre de sommets pas encore traites


  for (int i = 1; i < instance.nb_sommets; i++)
  {
    x = T[i - 1]; // Sommet de départ
    // Suppression du sommet dans la liste
    L[Px] = L[nr]; // On retire l'ancien sommet de la liste.

    for (int i = 0; i < 5; i++) // Initialisation du tableau des plus proches voisins
      V[i] = -1; // Pas de proche voisin au démarrage

    for (int j = 0; j < nr; j++) // Pour chaque sommet restant (rappel : on a retiré x)
    {
      y = L[j]; // Sommet actuel
      for (int k = 0; k < 5; k++) // On essaire de l'insérer dans les plus proches voisins
      {
        if (V[k] == -1 || instance.distances[x][y] < instance.distances[x][V[k]]) // Si la distance est plus courte, on insere le sommet
        {
          // décalage de tous les voisins plus grands sur la droite
          decalage_droite(V, k);
          decalage_droite(pV, k);
          V[k] = y; // insertion de l'élément choisi
          pV[k] = j;
        }

      }
    }
```

FIGURE 4 – PPVR - boucle de choix dans les 5 plus proches voisins

```
    // Initialisation pour la boucle de choix du voisin
    int k = 0;
    bool stop = false;

    // Boucle de choix du voisin
    while (!stop)
    {
      int r = rand() % 100;

      if (r < 80)
      {
        T[i] = V[k];
        Px = pV[k];
        stop = true;
      } else
      {
        k++;
        if (k > 6)
        {
          T[i] = V[4];
          Px = pV[4];
          stop = true;
        }
      }

    }
    nr--;
  }
}
```

3 Grasp

Lorsqu'on obtient une solution à partir d'un tour géant (cf. SPLIT), on obtient un certain nombre de tournées avec chacune un véhicule et un coût. La somme de ces coûts donnent le coût total de la solution.

Le but du grasp est d'améliorer une solution, pour ça on se sert de différents *opérateurs* qui peuvent améliorer chacun la solution avec des manières différentes. Lorsque un *opérateurs* améliore la solution, sa probabilité d'etre utilisé dans le grasp augmente, et inversement si il échoue sa probabilité augmente.

3.1 Opérateur 2OPT

3.1.1 Principe

Cet opérateur consiste à parcourir chaque tournée et essayer d'inverser l'ordre de deux sommets dans cette tournée. Si cette inversion est possible (le volume de la tournée est transportable par le véhicule) et qu'elle améliore le coût, alors on l'effectue.

3.1.2 Algorithme

```
1  pour i = 0 à nb_tournees faire
2      T ← Tournées[i];
3      pour a = 1 à T.nb_sommets − 3 faire
4          pour b = a à T.nb_sommets − 2 faire
5              x ← T[a];
6              y ← T[b];
7              m ← Distance(x − 1, x) + Distance(x, y) + Distance(y, y + 1);
8              p ← Distance(x − 1, y) + Distance(y, x) + Distance(x, y + 1);
9              si m > p alors
10                 Echange_Entre(a, b);
11                 Mettre à jour le cout de la tournée;
12                 Mettre à jour le cout de la solution;
13                 retourner Vrai;
14             fin
15         fin
16     fin
17 fin
18 retourner Faux;
```

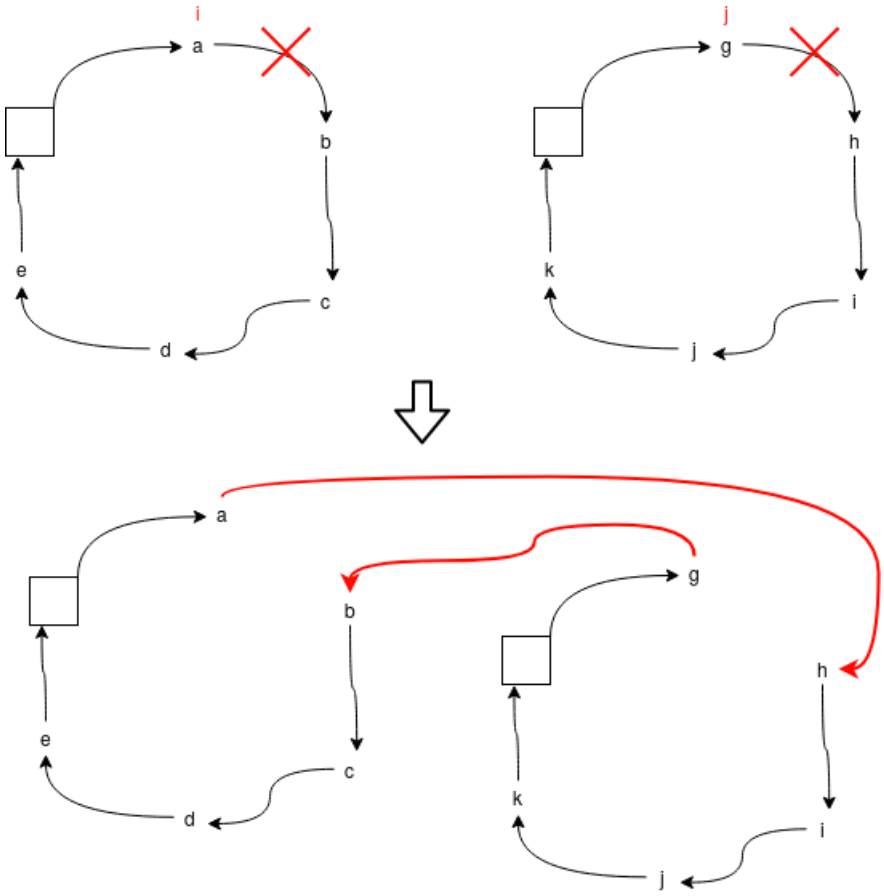
Algorithme 1 : Algorithme 2OPT

3.2 Opérateur 2OPT inter-tournée

3.2.1 Principe

Cet opérateur consiste à parcourir les tournées d'une solution et à chaque itération, en avoir deux pour tester de transformer ces deux tournées. Le but est de parcourir les sommets de ces deux tournées et d'essayer, lorsque ça diminue le coût, de transformer la 1ere tournée avec son début et la fin de l'autre tournée et inversement.

FIGURE 5 – Représentation graphique de l'opérateur 2OPT inter-tournée



3.2.2 Algorithme

```
1  pour i = 0 à nb_tournees − 2 faire
2      T1 ← Tournées[i];
3      pour j = i + 1 à nb_tournees − 1 faire
4          debut_t1 ← = 0;
5          T2 ← Tournées[j];
6          pour a = 1 à T1.nb_sommets − 2 faire
7              debut_t2 ← = 0;
8              debut_t1 += Distance(T1[a − 1], T1[a]);
9              pour b = 1 à T2.nb_sommets − 2 faire
10                 debut_t1 += Distance(T1[b − 1], T2[b]);
11                 passage_a_b ← Distance(T1[a], T2[b + 1]);
12                 passage_b_a ← Distance(T1[b], T2[a + 1]);
13                 fin_t1 ← T1.distance - debut_t1 - Distance(T1[a], T1[a + 1]);
14                 fin_t2 ← T2.distance - debut_t2 - Distance(T2[b], T2[b + 1]);
15                 distance_total_t1 ← debut_t1 + passage_a_b + fin_t2;
16                 distance_total_t2 ← debut_t2 + passage_b_a + fin_t1;
17                 si distance_total_t1 + distance_totatl_t2 < T1.distance + T2.distnce alors
18                     Echange_A_Partir_De(a, b);
19                     Mettre à jour le cout des tournée;
20                     Mettre à jour le cout de la solution;
21                     retourner Vrai;
22                 fin
23             fin
24         fin
25     fin
26 fin
27 retourner Faux;
```

Algorithme 2 : Algorithme 2OPT inter-tournée

3.3 Opérateur insertion

3.3.1 Principe

Cet opérateur consiste à parcourir les tournées d'une solution et de prendre 2 tournées de la même façon que l'opérateur 2OPT inter-tournée. La différence est que lui essaye de retirer un sommet de la première tournée et de l'insérer dans la deuxième à tous les endroits possibles.

3.3.2 Algorithme

```
pour i = 0 à nb_tournees − 1 faire
    T1 ← Tournées[i];
    pour j = 0 à nb_tournees − 1 faire
        si i ≠ j alors
            si T1.nb_sommets == 3 alors
                pour b = 0 à T2.nb_sommets − 1 faire
                    cout_t2 ← T2.cout + T2.cout_modulaire * (Distance(T2[b], T1[a]) + Distance(T1[a], T2[b+1]) -
                      Distance(T2[b], T2[b+1]));
                    si cout_t2 < T1.cout + T2.cout alors
                        Insertion(a);
                        Mettre à jour le cout de la tournée T2;
                        Suppression de la tournée T1;
                        Mettre à jour le cout de la solution;
                        retourner Vrai;
                    fin
                fin
            fin
            sinon
                pour a = 1 à T1.nb_sommets − 1 faire
                    pour b = 0 à T2.nb_sommets − 1 faire
                        m ← Distance(T1[a−1], T1[a]) + Distance(T1[a], T1[a+1]) + Distance(T2[b], T2[b+1]);
                        p ← Distance(T2[b], T1[a]) + Distance(T1[a], T2[b+1]) + Distance(T1[a−1], T1[a+1]);
                        si m > p alors
                            Insertion(a);
                            Mettre à jour le cout des tournée;
                            Mettre à jour le cout de la solution;
                            retourner Vrai;
                        fin
                    fin
                fin
            fin
        fin
    fin
fin
retourner Faux;
```

Algorithme 3 : Algorithme OPT Insertion