Université Clermont Auvergne

Institut Supérieur d'Informatique, de Modélisation
et de leurs Applications

Recherche Opérationnelle

HVPR

Réalisé par :
SAIDI Yasmine

Encadré par :
MR. Philippe Lacomme

Année universitaire 2020/2021

Introduction

Le problème HVRP (Heterogeonous Vehicle Routing Problem) est un problème d'optimisation combinatoire et de recherche opérationnelle. Il fait partie de la catégorie des problèmes de transport. E n fait c'est une variante du problème de VRP (Vehicle Routing Problem).

Un problème de HVRP contient un ensemble de clients qui ont chacun une demande. Pour effectuer la distributionn on dispose d'un nombre de véhicules de types différents. Chaque type de véhivule a une capacité de chargement, un coût fixe et un coût variable pour chaque kilomètre. L'objectif est de trouver un ensemble de tournées qui permettent de passer par toutes les villes en respectant les contraintes du problème (capacité et nombre de véhicules) et avec un coût minimal.

Dans de ce TP, on va s'intéresser à l'implémentation de l'algorithme d'optimisation d'un HVRP.

1 L'optimisation d'un HVRP

1.1 Génération d'un graphe

Les données de ce tp étaient sur http ://fc.isima.fr/ lacomme/hvrp/hvrp.html . Pour chaque instance, on trouve une première ligne qui contient le nombre de clients et le nombre types de véhicules. Puis, une ligne est dédiée pour chaque type de véhicule décrivant le nombre de véhicules dans chaque type, leurs capacités, leurs coûts fixes et leurs coût variables . Puis, on trouve une matrice qui représente les distances entre chaque deux clients. En fin, il y'a une liste des demandes de chaque client. Prenant comme exemple l'instance de puy de dome.

```
173 5
5  100 100 2.5
5 500 80 5.2
5 1000 180 6.2
5 1500 120 9.2
5 3000 120 9.2
0       87804   77276   116814  5023    5008
36206   60547   33016   79067   56412   39168
29384   10285   80058   50619   28132   81813
82196   91790   25031   95696   9794    42821
91721   32769   55407   32335   48011   42973
20410   78298   28061   18739   64529   25355
35941   18890   47106   29480   11303   46213
51270   36740   48329   26688   25716   69590
40980   34019   121690  32732   8738    95905
54891   15681   17064   20081   45181   60344
```

FIGURE 1 – Première instance

Pour cette instance nous avons un problème de HVRP avec 173 clients et 5 types de véhicule. Les véhicules du premier type sont au nombre de 5 avec une capacité de 100, un coût fixe de 100 et un coût variable de 2.5.Le client 1 a une demande de 313.

Pour illustrer les étapes de résolution on va prendre l'exemple suivant :
Le nombre de clients est 5. La demande de chaque client est dans le tableau suivant :

1	2	3	4	5
8	4	3	9	1

Le nombre de types de véhicules est 2. Les caractéristiques de chaque véhicule sont dans les tableau suivant :

type véhicule	capacité	cout fixe	cout variable
1	10	5	4
2	30	3	5

le graphe qui illustre ce problème est le suivant :

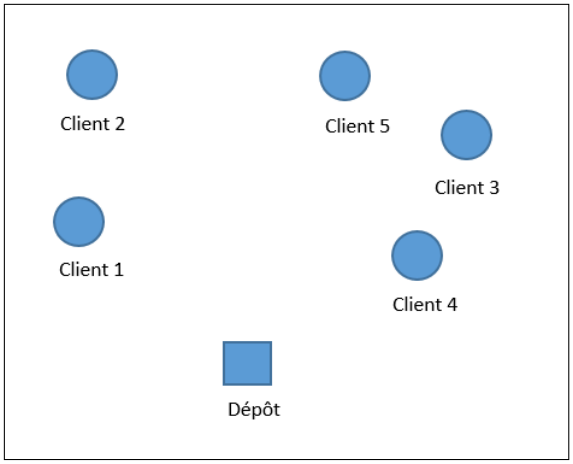

FIGURE 2 – Le graphe

1.2 Construire une solution initiale

Il existe plusieurs méthodes :

1.2.1 Plus proche voisin

On prend un noeaud, on cherche son plus proche voisin. Puis on cherche le plus proche voisin de ce dernier et ainsi de suite.

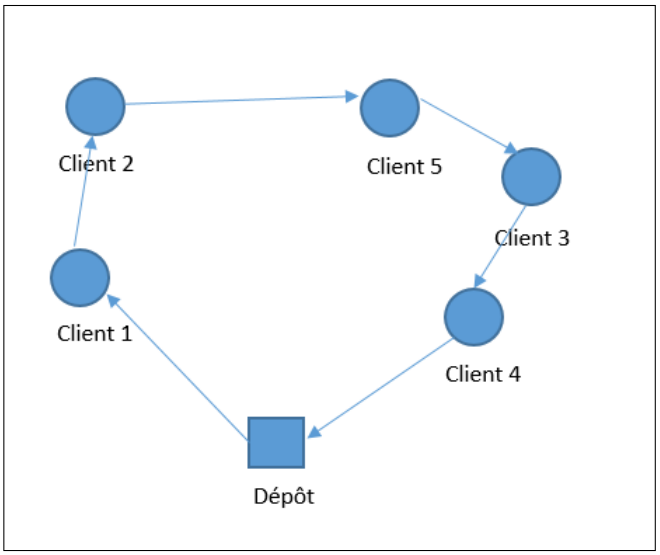

FIGURE 3 – Plus proche voisin

1.2.2 Plus proche voisin randomisé

On prend un sommet, on cherche les 5 plus proche voisin. Puis on choisit aléatoirement un sommet et on continue de la même manière.

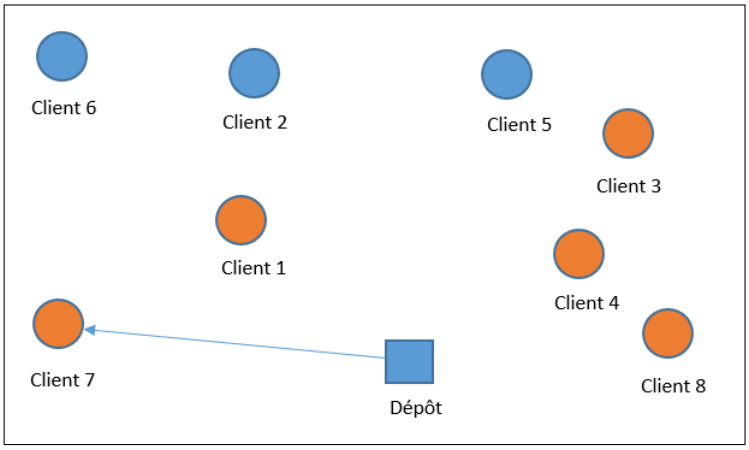

FIGURE 4 – Plus proche voisin randomisé

1.3 La recherche locale

La recherche locale consiste, une fois la solution initiale est construite, on peut l'améliorer en recherchant la meuilleure solution de son voisinage.

Nous avons travaillé avec 3 méthodes : 2-opt, le déplacement d'un sommet et le swap.

1.3.1 2-opt

2-opt est un algorithme de recherche locale à chaque étape, on supprime deux arrêtes de la solution courante et on reconnecte les deux tours formés. Cette méthode permet entre autres, d'améliorer le coût des solutions en supprimant les arêtes sécante.

cout(t2) = cout (t1) - d[2][1] - d[5][7] + d[2][5] + d[1][7]

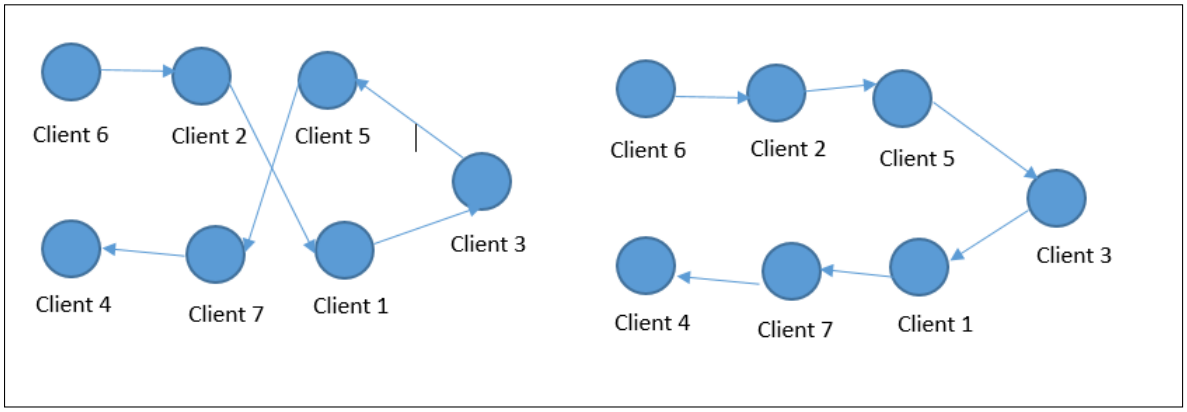

FIGURE 5 – 2-opt

Lorsqu'on inverse l'ordre de passage de deux villes, on inverse également l'ordre de passage de toutes les villes entre eux.

1.3.2 Déplacement d'un sommet

Cette méthode consiste à prendre un sommet et de le mettre entre deux autres sommets.

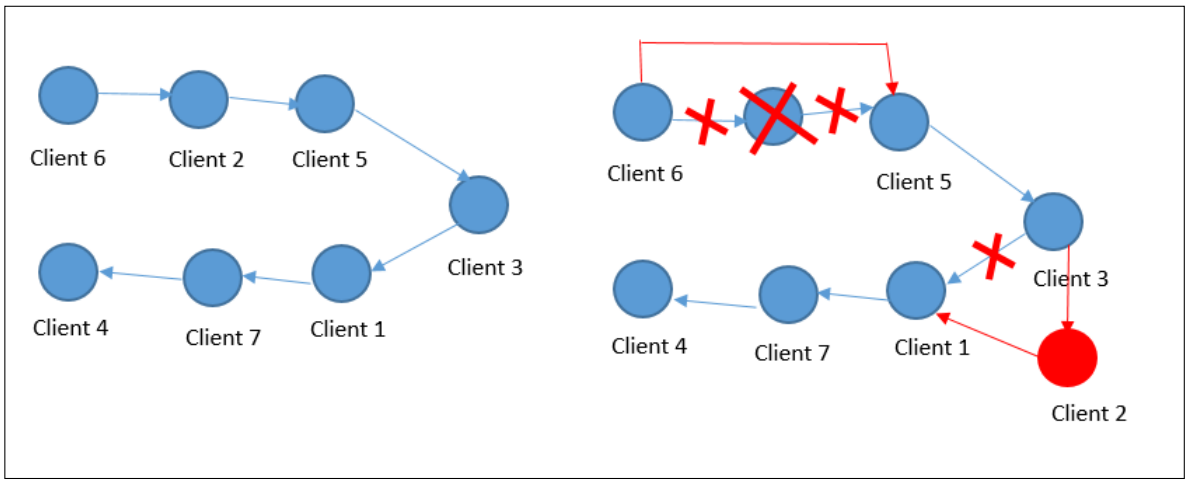

FIGURE 6 – Déplacement d'un sommet

1.3.3 Swap

Cette approche consiste à permuter deux clients dans le tour géant.

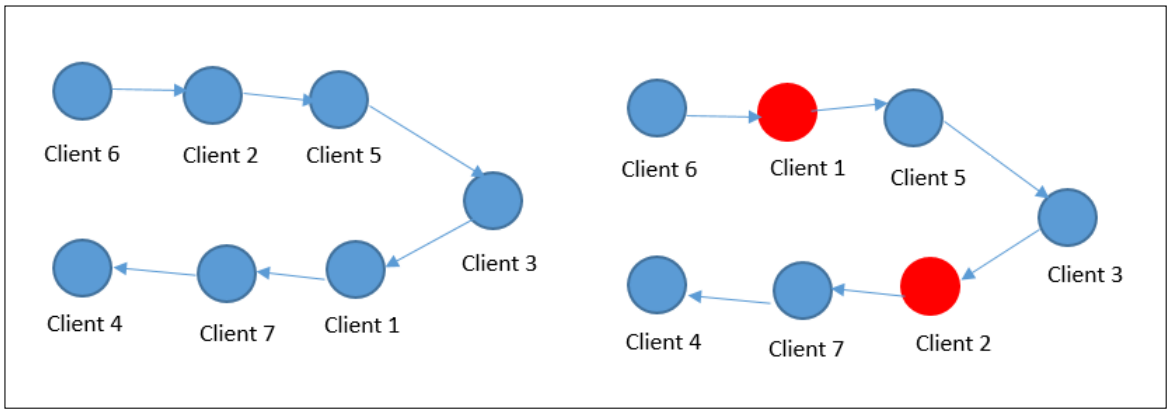

FIGURE 7 – Swap

1.4 Split

Le principe de Split est de découper le tour géant en des tournée réalisées par les différents véhicule que nous avons en respectant les contraintes de disponibilité des véhicules et de leurs capacités.

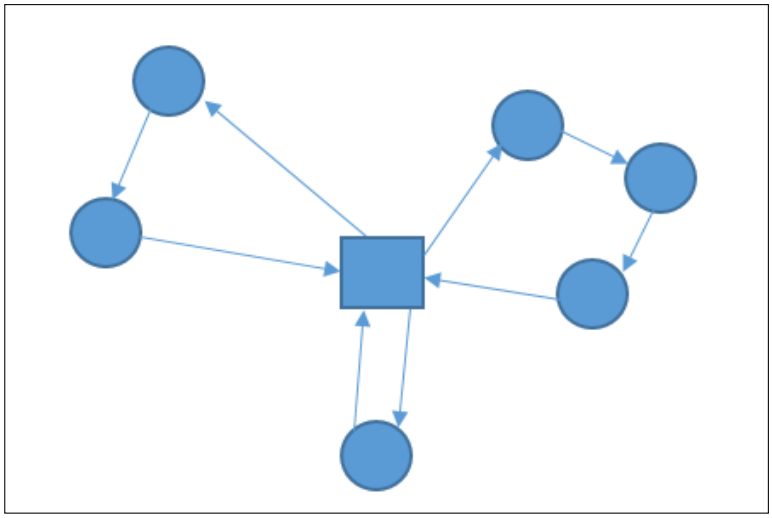

FIGURE 8 – Les tournées

2 Implémentation

2.1 Lecture d'un fichier

```
void lecture(string nom_fichier, T_instance& nom_instance) {
    ifstream fichier(nom_fichier, ifstream::in);
    if (fichier) {
        fichier >> nom_instance.nbre_client;
        fichier >> nom_instance.nbre_type_vehicule;
        for(int i=0 ; i<nom_instance.nbre_type_vehicule ; i++){
            fichier >> nom_instance.vehicule_par_type[i];
            fichier >> nom_instance.Capacite[i];
            fichier >> nom_instance.cout_fixe[i];
            fichier >> nom_instance.cout_variable[i];
        }
        for(int i=0 ; i<nom_instance.nbre_client+1 ; i++){
            for(int j=0 ; j<nom_instance.nbre_client+1 ; j++){
                fichier >> nom_instance.distance[i][j];
            }
        }
        for(int i=0 ; i<nom_instance.nbre_client ; i++){
            int client;
            fichier >> client;
            fichier >> nom_instance.demande[client];
        }
    }
    else {
        cout << "impossible d'ouvrir le fichier" << endl;
    }
    fichier.close();
}
```

FIGURE 9 – Lecture d'une instance

2.2 Plus proche voisin

```
void heuristique_plus_proche_voisin(T_instance& nom_instance,T_tour_geant& solution_initiale){
    int list_clt[nom_instance.nbre_client];
    for(int i=0;i<nom_instance.nbre_client;i++){
        list_clt[i]=i+1;
    }
    solution_initiale.list_clt[0]=0;
    solution_initiale.list_clt[nom_instance.nbre_client+1]=0;
    int size=nom_instance.nbre_client;
    int courant = 0;
    while(size!=0){
        int client_plus_proche=0;
        int distance_min=infini;
        int indice;
        for(int i=0;i<size;i++){
            if (nom_instance.distance[courant][list_clt[i]]<distance_min){
                distance_min=nom_instance.distance[courant][list_clt[i]];
                client_plus_proche=list_clt[i];
                indice=i;
            }
        }
        solution_initiale.list_clt[nom_instance.nbre_client-size+1]=client_plus_proche;
        solution_initiale.cout+=nom_instance.distance[courant][client_plus_proche];
        courant=client_plus_proche;
        size--;
        list_clt[indice]=list_clt[size];
    }
    solution_initiale.cout+=nom_instance.distance[courant][0];
}
```

FIGURE 10 – Implémentation du plus proche voisin

2.3 Plus proche voisin randomisé

```
void heuristique_plus_proche_voisin_randomise(T_instance& nom_instance,T_tour_geant& solution_initiale){
    int list_clt[nom_instance.nbre_client];
    for(int i=0;i<nom_instance.nbre_client;i++){
        list_clt[i]=i+1;
    }
    solution_initiale.list_clt[0]=0;
    solution_initiale.list_clt[nom_instance.nbre_client+1]=0;
    int size=nom_instance.nbre_client;
    int courant = 0;
    while(size!=0){
        int indice,indicee,nbre_voisin=0;
        int voisin[nb_voisin];
        int list_clt_copy[size];
        int size_copy=size;
        int client_plus_proche=0;
        for(int i=0;i<size_copy;i++){
            list_clt_copy[i]=list_clt[i];
        }
        while(nbre_voisin<nb_voisin and size_copy!=0){
            int distance_min=infini;
            for(int i=0;i<size_copy;i++){
                if (nom_instance.distance[courant][list_clt_copy[i]]<distance_min){
                    distance_min=nom_instance.distance[courant][list_clt_copy[i]];
                    client_plus_proche=list_clt_copy[i];
                    indice=i;
                }
            }
            voisin[nbre_voisin]=client_plus_proche;
            nbre_voisin++;
            size_copy--;
            list_clt_copy[indice]=list_clt_copy[size_copy];
        }
        int aleatoire = rand()%nb_voisin;
        solution_initiale.list_clt[nom_instance.nbre_client-size+1]=voisin[aleatoire];
        solution_initiale.cout+=nom_instance.distance[courant][voisin[aleatoire]];
        courant=voisin[aleatoire];
        //find index
        for (int i=0;i<size;i++){
            if (list_clt[i]==courant){indicee=i;}
        }
        size--;
        list_clt[indicee]=list_clt[size];
    }
    solution_initiale.cout+=nom_instance.distance[courant][0];
}
```

FIGURE 11 – Implémentation du plus proche voisin randomisé

2.4 Troisième plus proche voisin

```
void heuristique_troisieme_plus_proche_voisin(T_instance& nom_instance,T_tour_geant& solution_initiale){
    int list_clt[nom_instance.nbre_client];
    for(int i=0;i<nom_instance.nbre_client;i++){
        list_clt[i]=i+1;
    }
    solution_initiale.list_clt[0]=0;
    solution_initiale.list_clt[nom_instance.nbre_client+1]=0;
    int size=nom_instance.nbre_client;
    int courant = 0;
    while(size!=0){
        int indice,indicee;
        int list_clt_copy[size];
        int size_copy=size;
        int client_plus_proche=0;
        for(int i=0;i<size_copy;i++){
            list_clt_copy[i]=list_clt[i];
        }
        for (int j=0;j<3;j++){
            int distance_min=infini;
            for(int i=0;i<size_copy;i++){
                if (nom_instance.distance[courant][list_clt_copy[i]]<distance_min){
                    distance_min=nom_instance.distance[courant][list_clt_copy[i]];
                    client_plus_proche=list_clt_copy[i];
                    indice=i;
                }
            }
            size_copy--;
            list_clt_copy[indice]=list_clt_copy[size_copy];
        }
        solution_initiale.list_clt[nom_instance.nbre_client-size+1]=client_plus_proche;
        solution_initiale.cout+=nom_instance.distance[courant][client_plus_proche];
        courant=client_plus_proche;
        //find index
        for (int i=0;i<size;i++){
            if (list_clt[i]==courant){indicee=i;}
        }
        size--;
        list_clt[indicee]=list_clt[size];
    }
    solution_initiale.cout+=nom_instance.distance[courant][0];
}
```

FIGURE 12 – Implémentation du troisième plus proche voisin

2.5 2-opt

```
void _2_opt_intra_tournee(T_instance& nom_instance,T_tour_geant& solution_initiale,int nb_max){
    int j,diff;
    int l[nom_instance.nbre_client+2];
    int tmp,i=0,r=0;
    exit:
    while(i!=nom_instance.nbre_client-2 and r<nb_max){
        j=i+2;
        while(j!=nom_instance.nbre_client){
            diff=-nom_instance.distance[solution_initiale.list_clt[i]][solution_initiale.list_clt
            if(diff<0){
                solution_initiale.cout=solution_initiale.cout+diff;
                for(int k=0;k<i+1;k++){
                    l[k]=solution_initiale.list_clt[k];
                }
                l[i+1]=solution_initiale.list_clt[j];
                int h=j-1;
                for(int k=i+2;k<j+1;k++){
                    l[k]=solution_initiale.list_clt[h];
                    h--;
                }
                for(int k=j+1;k<nom_instance.nbre_client+2;k++){
                    l[k]=solution_initiale.list_clt[k];
                }
                for(int k=0;k<nom_instance.nbre_client+2;k++){
                    solution_initiale.list_clt[k]=l[k];
                }
                r++;
                i=0;
                goto exit;
            }
            j++;
        }
        i++;

    }
}
```

FIGURE 13 – Implémentation de 2-opt

2.6 Déplacement d'un sommet

```
void deplacement(T_instance& nom_instance,T_tour_geant& solution_initiale){
    int diff,ancien_diff;
    for(int i=1;i<nom_instance.nbre_client+1;i++){
        int delete1=nom_instance.distance[solution_initiale.list_clt[i-1]][solution_initiale.list_clt[i]];
        int delete2=nom_instance.distance[solution_initiale.list_clt[i]][solution_initiale.list_clt[i+1]] ;
        for(int j=1;j<nom_instance.nbre_client+1;j++){
            int delete3=nom_instance.distance[solution_initiale.list_clt[j]][solution_initiale.list_clt[j+1]];
            int add1=nom_instance.distance[solution_initiale.list_clt[j]][solution_initiale.list_clt[i]];
            int add2=nom_instance.distance[solution_initiale.list_clt[i]][solution_initiale.list_clt[j+1]];
            if(i!=j){
                diff = - delete1 - delete2 - delete3 + add1 + add2;
                if(diff<ancien_diff){
                    solution_initiale.cout=solution_initiale.cout+diff;
                    int client_choisi=solution_initiale.list_clt[i];
                    for (int k=i;k<j;k++){
                        solution_initiale.list_clt[i]=solution_initiale.list_clt[i+1];
                    }
                    solution_initiale.list_clt[j]=client_choisi;
                    ancien_diff=diff;
                }
            }
        }
    }
}
```

FIGURE 14 – Implémentation de déplacement d'un sommet

2.7 Swap

```
void swap(T_instance& nom_instance,T_tour_geant& solution_initiale){
    int inf, sup;
    do{
        inf=rand() % nom_instance.nbre_client + 1 ;
        sup=rand() % nom_instance.nbre_client + 1 ;
    }while(inf == sup);
    int inter;
    inter=solution_initiale.list_clt[inf];
    solution_initiale.list_clt[inf]=solution_initiale.list_clt[sup];
    solution_initiale.list_clt[sup]=inter;
}
```

FIGURE 15 – Implémentation de swap

2.8 Split

```
}
void split(T_instance& nom_instance, T_tour_geant& tournee,  T_solution& solution, T_list_label(&labels)[nbre_client_max + 1]){
    int j = 0;
    bool stop = false;
    T_label courant;
    T_label suivant;
    for (int i = 1; i < tournee.nbre_client - 1; i++) {
        j = i + 1;
        double charge = 0;
        int clt_i = tournee.list_clt[i];
        while (j < tournee.nbre_client + 1 && stop == false) {
            int clt_j = tournee.list_clt[j];
            int clt_j_1 = tournee.list_clt[j - 1];
            for (int k = 1; k < labels[clt_i].nbre_liste+1; k++) {
                courant = labels[clt_i].liste[k];
                for (int p = 1; p < nom_instance.nbre_type_vehicule + 1; p++) {
                    if (courant.vehicule_dispo[p] > 0) { ...

                    }

                }
            }
            j++;
        }
    }
}
```

FIGURE 16 – Implémentation de split

2.9 Grasp

```
}
void split(T_instance& nom_instance, T_tour_geant& tournee,  T_solution& solution, T_list_label(&labels)[nbre_client_max + 1]){
    int j = 0;
    bool stop = false;
    T_label courant;
    T_label suivant;
    for (int i = 1; i < tournee.nbre_client - 1; i++) {
        j = i + 1;
        double charge = 0;
        int clt_i = tournee.list_clt[i];
        while (j < tournee.nbre_client + 1 && stop == false) {
            int clt_j = tournee.list_clt[j];
            int clt_j_1 = tournee.list_clt[j - 1];
            for (int k = 1; k < labels[clt_i].nbre_liste+1; k++) {
                courant = labels[clt_i].liste[k];
                for (int p = 1; p < nom_instance.nbre_type_vehicule + 1; p++) {
                    if (courant.vehicule_dispo[p] > 0) { ...

                    }

                }
            }
            j++;
        }
    }
}
```

FIGURE 17 – Implémentation de split

Conclusion

Ce TP m'a permis d'implémenter l'optimisation du problème de transport HVRP qui est un problème NP-difficile et visualiser la pertinence de la solution basée sur GRASP.