



**Institut Supérieur d'Informatique
de Modélisation et de Leurs
Applications**

Campus des Cézeaux
24 avenue des Landais
63173 AUBIERE cedex

TP2 – d'outils d'aide à la décision
de 2^{ème} année Filière F3

TRAVAIL PRATIQUE2

Vehicle Routing Problem with Time Windows

Présenté par : Júlia COUTO

VilmarJefté RODRIGUES DE SOUSA

Introduction

Ce travail pratique a été proposé par la matière d'Outils d'Aide à la Décision de la 2^{ème} année du cours d'Informatique de l'ISIMA. Dont le but est faire l'optimisation du problème de véhicules avec fenêtre de temps (Vehicle Routing Problem with Time Windows - VRPTW).

Selon documentation donné dans le cours, le VRPTW consiste à définir un ensemble de routes de manière à satisfaire la demande d'un ensemble de n clients, où une flotte de véhicules est disponible dans un dépôt central, qui est noté comme 0 ou 1, pour effectuer ces livraisons. L'objectif principal est de minimiser le nombre de véhicules utilisés puis, en critère secondaire, la distance totale parcourue.

Pour trouver les bons résultats on va lire des coordonnées et des caractéristiques du dépôt et des clients d'un fichier ensuite on va faire trois types de heuristiques de constructions à savoir : plus-proche-voisin, Savings de Clarke & Wright et aléatoire. Après on construira trois heuristiques d'amélioration : 2-Opt* ; Or-Opt ; et Shift. Et pour fin on va faire le codage d'une métaheuristique (dans notre travail on utilisera l'algorithme génétique).

On a organisé ce rapport de la manière suivante : D'abord on va décrire le problème, ensuite on va définir les structures utilisées et la fonction de lecture. Ensuite, on discutera des heuristiques de construction, d'amélioration et de l'algorithme génétique. Après, on va discuter les résultats, et faire la conclusion.

1 - CARACTÉRISTIQUES DU PROBLÈME

Comme on a décrit, on va faire le codage du problème de tournées de véhicules avec fenêtre de temps (VRPTW). On considère que le problème a les suivantes caractéristiques :

- a) La flotte est homogène ;
- b) Chaque véhicule a la même capacité Q ;
- c) La demande d_i de chaque client $i = 1 \dots n$ doit être traitée en une seule fois et par un seul véhicule ;
- d) Chaque client $i = 1 \dots n$ possède une fenêtre de temps $[A_i, B_i]$ dont A_i est la heure de ouverture et B_i est la heure de fermeture. Donc le véhicule qui arrive avant de A_i doit attendre jusqu'à A_i pour commencer le traitement et aucune véhicule peut arriver après B_i .
- e) La livraison dure un temps t_i ;
- f) Le dépôt a , aussi, une fenêtre de temps $[A_0, B_0]$, où A_0 correspond à la date plus tôt de sortie d'un véhicule et B_0 la date plus tard qu'il peut arriver au dépôt.

2 – CODIFICATION DU PROBLÈME

Comme on a déjà décrit, pour faire la codification il faut lire les instances, faire les heuristiques de construction, faire les heuristiques d'amélioration et enfin faire la métaheuristique génétique. Pour commenter tous ces topics on va commencer par les structures qui sont la base de la solution du problème.

Structures

On va mettre le nom et ensuite les variables de chaque structure qu'on a utilisé et après on va faire un bref commentaire.

La première structure créée est la **T_ordre** qui a les variables *int tourne* et *intseq*. Cette structure va être le vecteur avec la ordre de clientes de chaque solution.

Ensuite on a la **T_sommet** qui a les variables *x*, *y*, *demande*, *date_tot* (qui est lo A_i de chaque client), *date_tard* (B_i de chaque client) et *duree* (t_i), tous variables sont en float. Cette structures stock les paramètres de chaque cliente.

La structure **T_instance** est la structures qui stock tous que a été lu dans le fichier, dans cette structure il y a *leint N* qui est la quantité de sommets (clients) dans la instance, *leintcapacite* qui représente la capacité de chaque véhicule, une *Liste* de **T_sommet** qui stock tous les renseignements des clients et pour fin un matrix *distance* de float qui stock les distances entres les clientes ($i - j$), avec i e j de 1 jusqu'à N .

La structure **T_cell** va stocker les informations d'arrive du véhicule à chaque client. Pour le faire elle a comme paramètres *intseq* et *float* : *arrive*, *debut*, *fin* et *folga*(différence entre date d'arrivée et date plus tard).

La prochaine structure est la **T_Tourne** que va être la representation des routes de chaque véhicule, pour stocker tous les informations de chaque route il faut savoir la quantité de cliente que le véhicule a fait le traitement en *intqte*, la kilométrage en *float Km*, la quantité de charge dans le véhicule et un liste avec les informations de chaque traitement en Liste de *T_cell*].

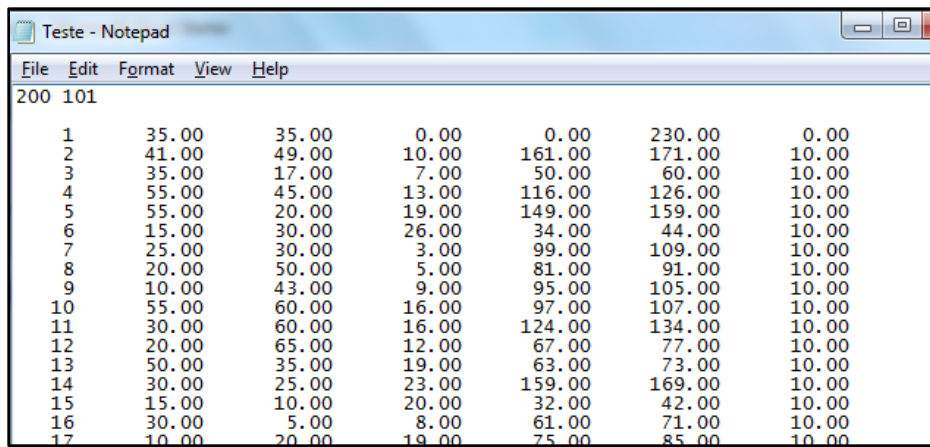
La **T_solution** est la structure qui représente la solution du problème, dans la solution est stocké : la quantité de tournes en *intqteTourne*, la kilométrage totale de cette solution en *floatKm_total*, l'ordre de clientes en *T_ordreListe[]*, les tournes en *T_TourneTournees[]* et enfin le cout de la solution en *float cout*.

La structure **T_population** est utile pour faire lamétaheuristique génétique et elle stock un vecteur avec 50 positions de **T_solution**.

Et pour fin, il y a la structure **T_gain**est utilisée pour stocker le gain en fusionnant deux tournées, une tournée qui commence par *int sommet1*, et une autre finit par *intsommet2*. Le *int* gstocke le gain de la fusion de les 2 tournées.

Lecture des instances

Les instances qu'on a prises pour faire tous les processus sont dans le site de Marius M. Solomon : <http://w.cba.neu.edu/~msolomon/problems.htm>. Pour faire la lecture on a copié les instances et les met dans une fichier comme la figure 1. On peut voir que le programme va lire d'abord la capacité des véhicules (200), la quantité de clients (101) et ensuite a chaque ligne il y a les informations de chaque client et du dépôt (ligne 0). Comme paramètre d'entrée, fournie par l'utilisateur, on a le nom du fichier contenant les données de la instance.



	1	2	3	4	5	6	7
200 101							
1	35.00	35.00	0.00	0.00	230.00	0.00	
2	41.00	49.00	10.00	161.00	171.00	10.00	
3	35.00	17.00	7.00	50.00	60.00	10.00	
4	55.00	45.00	13.00	116.00	126.00	10.00	
5	55.00	20.00	19.00	149.00	159.00	10.00	
6	15.00	30.00	26.00	34.00	44.00	10.00	
7	25.00	30.00	3.00	99.00	109.00	10.00	
8	20.00	50.00	5.00	81.00	91.00	10.00	
9	10.00	43.00	9.00	95.00	105.00	10.00	
10	55.00	60.00	16.00	97.00	107.00	10.00	
11	30.00	60.00	16.00	124.00	134.00	10.00	
12	20.00	65.00	12.00	67.00	77.00	10.00	
13	50.00	35.00	19.00	63.00	73.00	10.00	
14	30.00	25.00	23.00	159.00	169.00	10.00	
15	15.00	10.00	20.00	32.00	42.00	10.00	
16	30.00	5.00	8.00	61.00	71.00	10.00	
17	10.00	20.00	19.00	75.00	85.00	10.00	

Figure 1 – fichier pour lecture

Chaque ligne est divisé en 7 colonnes et chaque colonne représente un donné du client, ainsi la colonne: (1) est le numéro ; (2) est la coordonne X ; (3) est la coordonne Y ; (4) est la demande ; (5) est la date de ouverture ; (6) est la date de fermeture ; (7) est le temps de traitement.

Pour faire la lecture de ce fichier on a créé une fonction que s'appelle **lire_instance(T_instance &instance, String^ nom)**, où le paramètre *nom* est le nom du fichier, qui a été donné par le l'utilisateur, et le paramètre *instance* va être modifié avec les donnes du fichier. Pour modifier l'instance d'abord on a ouvré le fichier comme lecture, après on a mis capacité dans *instance.capacite* et quantité de cliente en *intance.N* ; ensuite la fonction entre dans un «for» que va lire chaque ligne du fichier et mettre ces valeur dans *intance.Liste[i]* (*i* = 1..*instance.N*). Après la lectures de tous les clients la fonctions va calculer la distance entre le cliente *i* e le client *j* (*i,j* = 1..*instance.N*) pour la formula : $dist = \sqrt{(X_i - X_j)^2 + (Y_i - Y_j)^2}$.

Après le calcul de tous les distances la fonction est finie.

Les Heuristique de Construction

On a fait trois types de heuristique de constructions : plus-proche-voisin , Savings de Clarke &Whight, et aléatoire. Le but de cette approche est pleine la structure **T_population** avec 50 solutions qui iront à l'algorithme génétique. Pour compléter les 50 positions on va mettre dans les 20 premières positions les solutions de plus proche voisin, ensuite de la 21^{ème} jusqu'à 40^{ème} position on va mettre les solutions de Savings de Clarke &Whight , enfin les 10 dernières positions on va mettre les solutions totalement aléatoires.

Avec cette approche on va obtenir une grandediversité de solutions pour faire l'algorithme génétique. Il faut observer que dans toutes les constructions il y a la caractéristique aléatoire, puis on a besoin de résultats différents à chaque itération. Suive les explications et les algorithmes des trois types de construction qu'on a fait le codage.

1) Plus-proche-Voisin :

Dans notre algorithme le plus proche voisin est le clint qui a la moins grand distance en kilomètres du dernière client qui a été mis dans le vecteur solution. L'algorithme dans notre travail s'appelle **generer_ordre_distance(T_instance&instance, T_solution&solution)**. Comme on a dit, tous les algorithmes de construction ont la caractéristique aléatoire, donc dans l'algorithme du plus-proche-voisin on va mettre tous les voisins du client *i* dans un vecteur ordonnée et ainsi l'algorithme va donner des probabilités pour chaque voisin. On a mettre la probabilité initiale de 80%, ainsi l'algorithme va choisir par hasard un numéro entre 0 et 9 et le premier voisins (qui est vraiment le plus proche) va avoir probabilité de 80% d'être choisi, sinon

l'algorithme va prendre autre numéro entre 0 et 9 et donner la même probabilité pour le proche voisin (seconde, troisième...). L'algorithme va faire ça jusqu'à ce qu'un client soit choisi.

Puis que tous les voisins sont dans le vecteur ordre de la solution, l'algorithme commence à faire les routes dans la fonction évaluer laquelle s'appelle **generer_sol(T_solution &solution, T_instance&instance, int type)**, auquel le *type* 1 se réfère à la construction plus proche voisin. Cette fonction va mettre les clients d'accord avec l'ordre de la solution dans les routes, si un certain client ne peut pas être dans la route recourant (restrictions de la fenêtre de temps ou capacité du véhicule), l'algorithme va ouvrir une nouvelle route pour mettre ce nouveau client.

Suive le codage qui a été réalisé :

```
void generer_ordre_distance(T_instance &instance, T_solution &solution){
    Déclarations(c_ordre[n_max], disp[n_max], actuel= 1, find2, sort, a =1,
    k =1, disponible = instance.N - 1 )

    Initialisations(disp[i] = 0 pour tout chaque i, i<= instance.N)

    alors que(k<instance.N){
        // remplir vecteur c_ordre avec les sommets dans un ordre du plus proche de actuel
    }
Alors que(find2==0){
    sort = (rand()%10);
    si(sort < 8){ //Une probabilité de 80% pour choisir chaque sommet
        actuel=c_ordre[a];
        solution.ordre[k].seq = actuel;
        disponible = disponible - 1;
        k = k+1;
        disp[actuel] = 0;
        find2 = 1;
    }
    sinon{
        si(a < disponible){ a++;}
        sinon{a =1;}
    }
}
}
```

```
void generer_sol(T_solution &solution, T_instance &instance, int type){
    si(type == 1){
        generer_ordre_distance(instance, solution);
    }
    sinon(type == 2){
        generer_ordre_aleatoire(solution);
    }

    Déclarations(alloue = 1, insere)

    initialiser_tourne(instance, solution);
    alors que(alloue < instance.N){
        insere = insertion(instance, solution, solution.qteToune,
        solution.ordre[alloue].seq);
        si(insere == 0){
            new_tourne(instance, solution, solution.ordre[alloue].seq);
            alloue = alloue + 1;
        }
        sinon{
            // insérer alloue dans la tournée courante
            alloue = alloue + 1;
        }
    }
    solution.cout = alpha*solution.qteToune + beta*solution.Km_total;
    //le calcul du coût en fonction des paramètres d'entrée alpha e beta
}
}
```

2) Savings de Clarke & Whight

Selon document que a été donné dans le cours d'Outils, l'heuristique Savings de Clarke & Whight a été développée pour le Vehicle routing problem (VRP). Mais le document décrit que elle ne change pas beaucoup

pour le VRPTW, mis à part le teste de faisabilité temporelle. Pour faire cette heuristique on va crée initialement une tournées pour chaque client. Pour tout arc (i,j), elle calcule le gain $g_{ij} = c_{i0} + c_{0j} - c_{ij}$, auquel le « c_{ij} » est le cout de distance pour aller du client i pour client j ; alors g_{ij} est le gagne d'assemble deux tournes.

On choisit alors un gain dans le vecteur, chaque gain a une probabilité de 80% d'être choisi, ce qui assure une petite aléatoire à la solution.Elle vérifiera s'il ya les tournées indiqués par le gain choisi, et si cestournées peuvent être fusionnées(en utilisant la fonction **tester_insertion()**). Si elles peuvent être fusionnées,elle fait la fusion avec la fonction **fusionner_tournee()**. Sinon, on passe au prochain gain.

Suive l'algorithmequ'on a fait pour réaliser le codage de la fonction :

```
void generer_sol_CW(T_solution &solution, T_instance &instance){

Déclarations(gain[n_max*n_max], sort, insere, utilise[n_max*n_max] )
Initialisations(utililise[i] = 0 pour tout chaque i, i<= instance.N)

pour (i = 2; i<=NT; i++)
pour(j=2; j<=NT; j++){
si(i!=j){
    gain[k].g = instance.distance[i][1]+ instance.distance[1][j] -
    instance.distance[i][j];
    //calculer le gain dans la fusion d'une tournée qui commence par et finit avec
    j et trier ce gain dans l'ordre décroissant
    K++
}
}
pour(i=2; i<=NT; i++){
    solution.qteToune ++;
    new_tourne(instance, solution, i);
}
alors que(util != (k-1)){
pour( i = 1; i<k; i++){
    sort = (rand()%10);
    si(sort < 8 && utilise[i]==0){ //Une probabilité de 80% pour choisir gain
si(//il y a une tournée qui commence par l et finit avec j){
        insere = tester_insertion(instance, solution, j, 1);
        si(insere == 1)
            fusionner_tournee(instance, solution, j, 1);
}
}
    utilise[i] = 1;
    util++;
}
} //le calcul du coût en fonction des paramètres d'entrée alpha e beta
solution.cout = alpha*solution.qteToune + beta*solution.Km_total;
} //fin fonction
```

3) Heuristique aléatoire

Cette heuristique est pareil à construction de plus-proche-voisin, mas ici la choisi des clientes pour mettre dans le vecteur ordre est totalement aléatoire, en utilisant la fonction**generer_ordre_aleatorie(T_solution &solution)**. Après le vecteur ordre est plein avec tous les clients, on va le renvoyer pour la fonction **generer_sol()**où l'algorithme construira les routes avec le vecteur de ordre de la solution.

Les Heuristique d'amélioration

L'idée principale est qu'à partir d'une solution, on va chercher des mouvements qui permettre de réduire la quantité de routes et la quantité de kilomètres des véhicules. Alors on va faire le codage des trois types d'amélioration que sont dans les spécifications du travail. Les trois types sont :**2-Opt***, **Or-Opt**, et **Shift**.

Comme on a vu en cours les deux premières sont de mouvements entre deux routes différents ainsi sont les mouvements dans lesquels est possible d’avoir des réductions du nombre de tournes, mais dans Shift il y a seulement les mouvements intra-tournes, ainsi il y a seulement de réduction de kilométrage dans chaque tournée.

1) 2-Opt*

Solent le cours d’outils ce mouvement consiste à changer les derniers clients de deux tournes. On peut voir la figure 2 pour comprendre mieux le fonctionnement du mouvement :

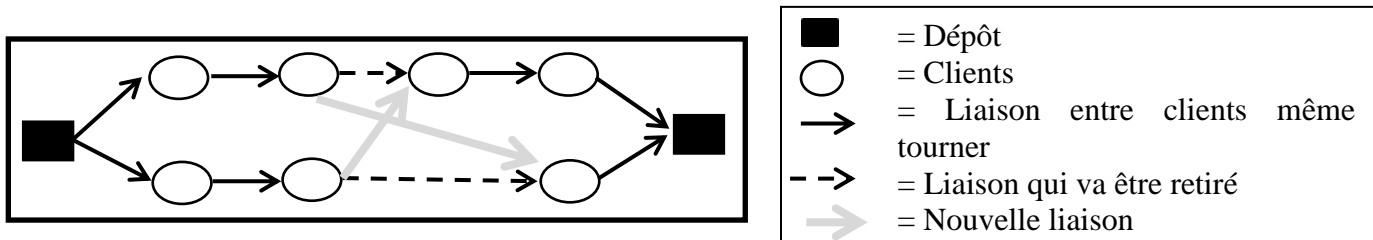


Figure 2 – Mouvement 2-Opt*

Pour faire le mouvement on a créé la fonction **recherche_deux_opt(T_instance&instance, T_solution &solution)** qui a comme paramètres l’instance du programme avec tous le données qui a été lus et la solution qui va être changé. Cette fonction échange les derniers sommets de deux tournes et si le resultat est mieux, elle modifie la solution. Suive le codage :

```
void recherche_deux_opt(T_instance &instance, T_solution &solution){

    Déclarations(temp, i=NT-1, j = i)

    alors que(i>1){
        j = i-1;
        alors que(j>0){
            si(solution.ordre[i].tourne!=solution.ordre[j].tourne){
                copie_solution(temp, solution);
                temp.ordre[j].seq = solution.ordre[i].seq;
                temp.ordre[i].seq = solution.ordre[j].seq;
                generer_sol(temp, instance, 0);
                si(temp.cout<solution.cout){
                    copie_solution(solution, temp);
                    i = NT-1;
                    j = i-1;
                }
                sinon{
                    alors que(solution.ordre[j].tourne == solution.
                    ordre[j-1].tourne && j>1){
                        j--;
                    }
                    j = j -1;
                }
            }
            sinon{j--;}
        }
        alors que(solution.ordre[i].tourne == solution.ordre[i-1].tourne && i>2){
            i--;
        }
        i = i - 1;
    }
}
```

2) Or-Opt

Solen document donné dans cours ce mouvement consiste à déplacer une séquence de clients dans une autre tournée. Dans notre travail l'algorithme va déplacer toujours seulement un client dans une autre tournée, elle prendra le premier client d'une tournée et va essayer le mettre en dans le moyen des toutes les autres tournées, si le mouvement est possible et améliorer la solution alors elle va accepter le changement. Pour faire le mouvement on a créé la fonction **recherche_or_opt(T_instance&instance, T_solution &solution)**

La figure 3 affiche un exemple de changement :

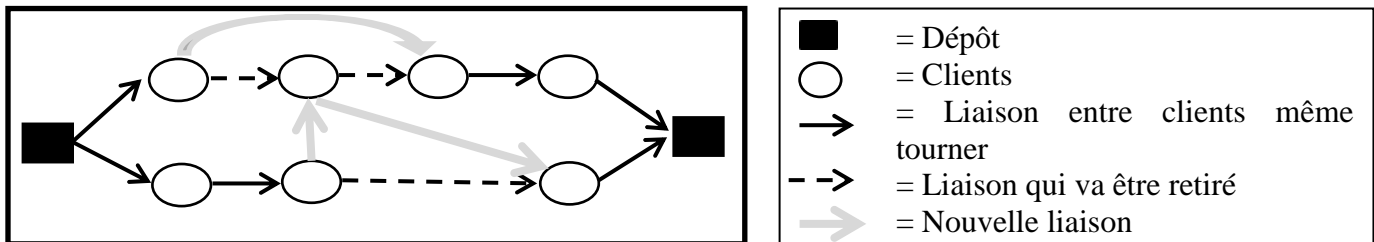


Figure 3 – mouvement Or-Opt

Suive le codage d'algorithme :

```
void recherche_or_opt(T_instance &instance, T_solution &solution){

    Déclarations(temp, i = 1, j = i)

    alors que (i < NT-1) {
        j = i+1;
        alors que (j < NT) {
            si (solution.ordre[i].tourne != solution.ordre[j].tourne) {
                copie_solution(temp, solution);
                temp.ordre[j].seq = solution.ordre[i].seq;
                generer_sol(temp, instance, 0);
                si (temp.cout < solution.cout) {
                    copie_solution(solution, temp);
                    i = 1;
                    j = i+1;
                }
            }
            sinon {
                alors que (solution.ordre[j].tourne == solution.ordre[j+1].tourne
                && j < NT-2) {

                    j++;
                }
                j = j + 1;
            }
        }
        sinon {j++;}
    }
    alors que (solution.ordre[i].tourne == solution.ordre[i+1].tourne && i < NT-3) {
        i++;
    }
    i = i + 1;
}

}
```


3) Shift

Comme on a déjà décrit, le mouvement shift est un mouvement intra-tourne alors il permet de d'optimiser localement la tournée, ça est réduire la distance que le véhicule va parcourir pour faire le traitement de tous les clients de la tournée. Le principe est prendre chaque client dans la tournée et chercher le meilleur endroit pour le mettre dans la tournée.

La figure 4 choisi le mouvement du deuxième client pour le fin de la tournée.

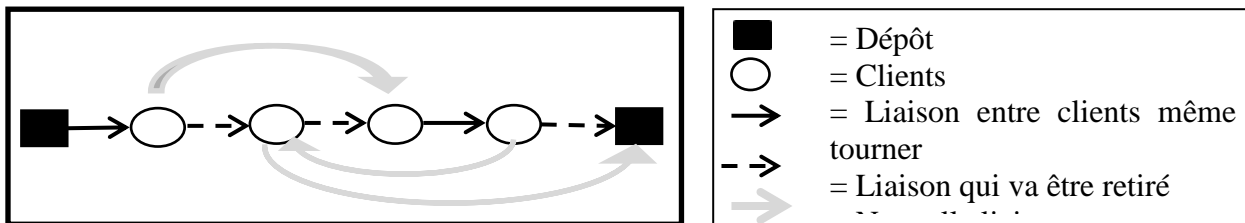


Figure 4 – Mouvement Shift

Ce mouvement est important, puis la distance est le critère secondaire de la optimisation du problème.

Pour faire ce mouvement on a créé la fonction **recherche_shift(T_instance &instance, T_solution &solution).**

Suive la codification de ce algorithme :

```
void recherche_shift(T_instance &instance, T_solution &solution){
    Déclarations(temp, i = 1, j = i + 1)

    alors que(i < NT-1){
        alors que(j < NT){
            si(solution.ordre[i].tourne == solution.ordre[j].tourne){
                copie_solution(temp, solution);
                temp.ordre[j].seq = solution.ordre[i].seq;
                temp.ordre[i].seq = solution.ordre[j].seq;
                generer_sol(temp, instance, 0);
                si(temp.cout < solution.cout){
                    copie_solution(solution, temp);
                    i = 1;
                    j = i+1;
                }
                sinon{j++;}
            }
            sinon{
                i++;
                j = i + 1;
            }
        }
        i++;
        j = i+1;
    }
}
```

La métaheuristique génétique

Selon Mitchell 1995, John Holland pendant les années 1960s a décrit l'algorithme génétique, l'idée est adapter la évolution naturel des plants et animaux pour système computationnelles. Les métaheuristique selon Mitchell [1995] est conseil pour problèmes qui ont besoin de solutions nouvelles et original sans perdre les caractéristiques de construction des parents.

Dans notre travail, toutes les solutions qu'étaient générées dans les trois types d'heuristiques de construction étaient cachées dans la structure T_Population. Cette structure va entrer dans la fonction **genetique(T_population &POP, T_instance&instance)** qui entre dans une itération de $i = 0$ jusqu'à *IntMAX*, lequel est entré par l'utilisateur sur l'écran d'accueil (« nombre d'itérations »). Dans cette itération l'algorithme va choisir par hasard une des 10 meilleures solutions du T_Population et une autre entre les 40 autres solutions, en utilisant la fonction **choisir_parent(int &P1, int &P2)**. Après elle générera une « fille » de ces deux solutions (fonction **croisement()**), cette fille va entrer dans le lieu du pire parent. Après le T_population va être trié et la itération va recommencer.

Pour générer la « fille » l'algorithme fait la copie de une partie du ordre de solution de un parent et après va faire mettre le reste d'accord avec le autre parent. La figure 5 affiche l'idée principal du algorithme.

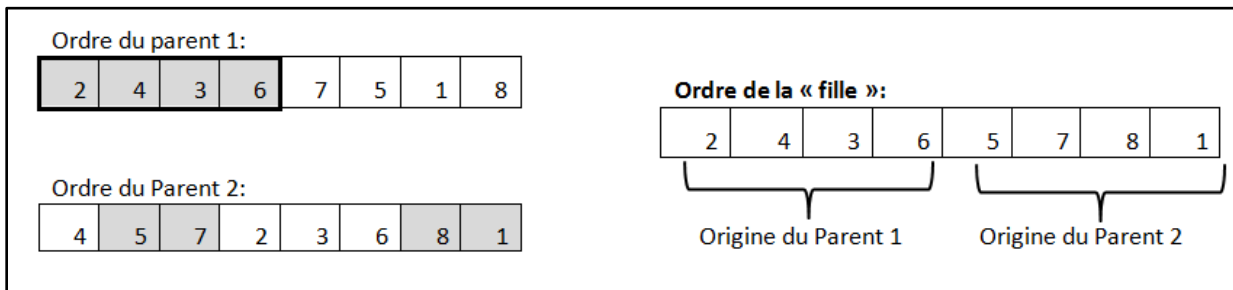


Figure 5 – Formation de nouvelle solution pour Algorithme génétique

Suive la codification du algorithme génétique :

```
void genetique(T_population &POP, T_instance &instance){
    Déclarations(Pere1, Pere2, new_sol, cont)

    alors que (cont<IntMAX){
        choisir_parent(Pere1, Pere2);
        croisement(POP.solutions[Pere1],POP.solutions[Pere2], new_sol);
        generer_sol(new_sol, instance, 0);
        recherche_deux_opt(instance, new_sol);
        recherche_or_opt(instance, new_sol);
        recherche_shift(instance, new_sol);
        copie_solution(POP.solutions[Pere1], new_sol);
        RETRIER(POP, Pere1);

        cont++;
    }
}
```

3 – RÉSULTATS

Pour faire générer les résultats on a utilisé une ordinateur : Acer; Model - AO722 ; Processeur - AMD C-60 APU avec Radeon (tm) HD Graphics 1.00 GHz ; RAM – 3.73GB ; System Type – 64 bit Operation System, System opérationnelle – Windows 7 Home Premium. Langage C++ en Visual Studio 2008.

Le tableau ci-dessous sert à comparer entre les valeurs trouvées dans notre travail et les meilleures solutions connues identifiées par les heuristiques. Les paramètres ont été utilisés sont les suivants: coût par tournée = 1000, coût par Kilométrage = 1, nombre de procédures génétiques = 300.

Problème	Kilométrage	Nombre de Tournées	Kilométrage optimal	Nombre de Tournées optimal
R101.25	644,40	8	617.1	8
R101.50	1198,52	14	1044.0	12
R101.100	2048,51	27	1637.7	20
R105.25	605,55	6	530.5	6
R105.50	1033,59	12	899.3	9
R105.100	1725,05	22	1355.3	15
R109.25	453,26	5	441.3	5
R109.50	885,14	10	786.8	8
R109.100	1465,69	17	1146.9	13
Problème	Kilométrage	Nombre de Tournées	Kilométrage optimal	Nombre de Tournées optimal
C101.25	240,92	4	191.3	3
C101.50	621,91	8	362.4	5
C101.100	1673,25	17	827.3	10
C105.25	253,40	4	191.3	3
C105.50	598,01	8	362.4	5
C105.100	1539,05	16	827.3	10
C109.25	228,66	3	191.3	3
C109.50	481,52	6	362.4	5
C109.100	1219,78	13	827.3	10
Problème	Kilométrage	Nombre de Tournées	Kilométrage optimal	Nombre de Tournées optimal
RC101.25	534,74	6	461.1	4
RC101.50	1076,48	11	944	8
RC101.100	2080,70	23	1619.8	15
RC105.25	470,67	5	411.3	4
RC105.50	1056,96	10	855.3	8
RC105.100	1873,36	20	1513.7	15
RC108.25	294,99	3	294.5	3
RC108.50	779,91	6	598.1	6
RC108.100	1362,38	11	1114.2	11

Figure 6 - Tableau des résultats et des comparaisons

On peut voir que les meilleurs résultats ont été trouvés pour les petites instances. Certains résultats étaient très proches ou égales aux valeurs optimales, alors que d'autres étaient beaucoup pire que prévu, en particulier dans les instances avec 100 sommets (clients).

Variations dans le fonctionnement du programme

On va faire maintenant une comparaison entre les résultats avec et sans l'utilisation de fonctions d'amélioration. On va utiliser l'instance R101, 100 clients et 300 procédures génétiques. Les résultats sont les suivants :

Figure 7 – À gauche sont les résultats avec les fonctions d'amélioration et à droite sont les résultats sans leur utilisation

On peut voir que l'utilisation de fonctions d'amélioration génère un gain important, notamment dans le nombre de véhicules utilisés, qui est le principal critère à analyser. Cependant, le temps d'exécution du programme a augmenté considérablement, étant approximativement 4 fois plus élevée. Cela est dû à la grande complexité des fonctions d'amélioration, et son utilisation encore et encore.

Maintenant, on va analyser l'amélioration générée par l'algorithme génétique.

Figure 8 - À gauche sont les résultats avec l'algorithme génétique et à droite sont les résultats sans lui utilisation.

On peut voir que, encore une fois, il y a une amélioration du nombre de véhicules utilisés. Il ya également une grande différence de temps de fonctionnement, le temps en utilisant 300 fois l'algorithme génétique est 2 fois plus grand que le temps sans l'utiliser.

Pour la fonction objectif, différentes méthodes de calcul du coût peut être utilisé. Dans les exemples précédents, on a utilisé une fonction de coût pour lequel il a été attribué un coût de 1000 euros pour chaque véhicule utilisé et un coût de 1 euro par kilomètre parcouru. Ainsi, on priorise le nombre de véhicules, et on analyse le kilométrage total comme critère secondaire.

Une autre forme d'analyse serait d'utiliser le kilométrage comme seul critère utilisé. Dans ce cas, on a les résultats suivants:

Figure 9 – À gauche sont les résultats qui privilégient la quantité de véhicules et à droite sont les résultats qui privilégient la kilométrage

Comme prévu, dans le premier cas, on a un plus petit nombre de visites et un kilométrage plus élevé, et dans le deuxième cas, le résultat opposé.

Une autre forme possible de la fonction objectif considère le coût comme une combinaison du nombre de véhicules et le kilométrage, en utilisant les multiplicateurs α et $(1 - \alpha)$.

4 – CONCLUSION

Le travail nous a permis de constater l'importance d'avoir les bonnes méthodes de construction et d'amélioration dans les résultats du problème. Nous avons également vu comment une variété de solutions de différentes qualités, peut être utile pour générer une nouvelle solution meilleure que les précédentes, en utilisant la procédure de croisement.

Un autre aspect intéressant a été vu comment le choix de la fonction objectif influe sur les résultats finaux. Le choix des paramètres de la fonction objectif doit être très prudent, en fonction de chaque situation.

Bibliographie

- [1] Marium M. Solom, 1984 – « Algorithms for the vehicles routing and scheduling problems whit window Constraints »
- [2] Melanie Mitchell, 1995 “Genetic Algorithms: An Overview”, Santa Fe Institute