جامعة السلطان مولاي سليمان
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵙⵙⵓⵍⵟⴰⵏ ⵎⵓⵍⴰⵢ ⵙⵍⵉⵎⴰⵏ
Université Sultan Moulay Slimane

ENSA KHOURIBGA ÉCOLE NATIONALE DES SCIENCES APPLIQUÉES

Rapport du travaux pratique METAHEURISTIQUEQ

Réalisé par :
ENNOURI ABDELOUAHED

Encadré par :
Mr HAFIDI IMAD

Année universitaire 2019/2020

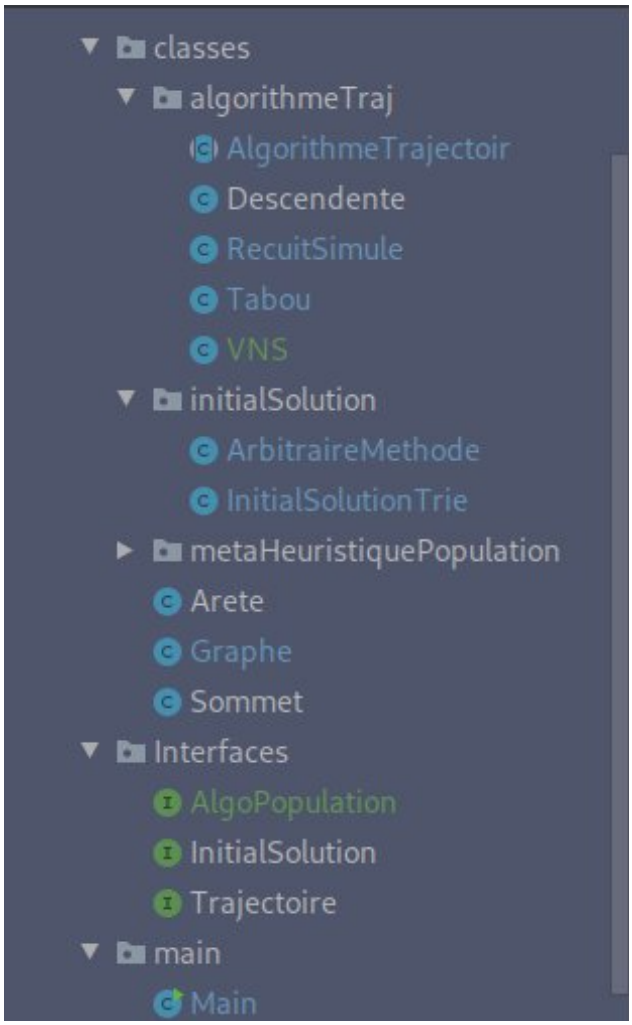
I - Introduction :

Le problème du voyageur de commerce, ou TSP pour Traveling-Salesman Problem, consiste, pour un graphe donne, de déterminer un cycle hamiltonien dont la longueur est minimale.Pas juste des villes et des distances : le TSP peut se rencontrer dans d'autres contextes, et/ou comme sous-problème : problèmes de logistique, de transport, d'ordonnancement, etc.

2-Génération d'un solution initiale :

Pour la génération d'un solution initiale on va utiliser deux méthode, la première qui se base dur le trie des arêtes et après choisir le chemins le plus court, et la deuxième méthode qui se base de choisir une sommet aléatoire et voir la sommet la plus proche pour construire une arête.

Structure général du projet :



Méthode de trie :

L'interface InitialeSolution est une interface fonction qui a une seule méthode initialeSolution qui va retourner une list de sommets.

```
@FunctionalInterface
public interface InitialSolution {
     List<Sommet> initialeSolution(Graphe graphe);
}
```

La class InitialSolutionTrie qui implemente l'interface InitialSolution

```
public class InitialSolutionTrie implements InitialSolution {
    @Override
    public List<Sommet> initialeSolution(Graphe graphe) {
        List<Sommet> listVisited=new ArrayList<>();
        for (Arete ar : graphe.getSortedAret()) {
            if (ar.getDebut().isVisited() ||
ar.getVoisin().isVisited()) {
                continue;
            } else {
                choisirAret(listVisited, ar);
            }
        }
        return listVisited;
    }
```

Méthode de genration arbitraire :

La classe ArbitraireMethode qui implemente l'interface InitialSolution et il a comme attribut de classe une graphe,

```
public class ArbitraireMethode implements InitialSolution {

    private Graphe graphe;

    public ArbitraireMethode(Graphe graphe) {

        this.graphe=graphe;

    }

@Override

public List<Sommet> initialeSolution(Graphe graphe) {

    ArrayList<Sommet> sommets=new ArrayList<>();

    sommets.add(sommetDepart());

    Sommet plusProchVoisin;

    while (true) {

        if (sommets.size()<graphe.getNbVille()) {

     plusProchVoisin=plusProcheVoisin(sommets.get(sommets.size()-
1), sommets);

            sommets.add(plusProchVoisin);

        } else {

            break;

        }

    }

    sommets.add(sommets.get(0));

    return sommets;

}
```

Exécution :

```
/**************Matrice********************/
0       90      14      69      68      53      83      1       31      40
90      0       16      88      69      94      86      88      24      4
14      16      0       10      45      12      41      88      61      40
69      88      10      0       36      1       80      98      52      68
68      69      45      36      0       19      54      89      84      36
53      94      12      1       19      0       49      42      53      36
83      86      41      80      54      49      0       75      27      35
1       88      88      98      89      42      75      0       9       95
31      24      61      52      84      53      27      9       0       88
40      4       40      68      36      36      35      95      88      0

/***************Initial Solution******************/
le cout du solution initial 486
[0->, 7->, 3->, 5->, 1->, 9->, 8->, 2->, 4->, 6->]
```

La génération d'une graphe [10,10] avec une solution initiale avec la méthode du trie.

3- La méthode descente.

Algorithme utilisé :

- Choix de la solution initiale décrite plus haut (voisin le plus proche)
- Déterminer une solution voisine s' qui minimise f dans N(s)
- Si f(s')<f(s) alors poser s =s' et retourner à 2, jusqu'a ce que la solution s ne s'améliore plus, au bout de n itérations (n = nombre de villes)

La classe Descendente hérite de class abstract AlgorithmeTrajectoir

```
public class Descendente extends AlgorithmeTrajectoir {

    public Descendente(Graphe graphe) {

        super(graphe);

    }
```

La méthode solve retourne le chemin optimiser :

```java
@Override
public List<Sommet> solve(List<Sommet> initialSolution) {
    List<Sommet> sommets=initialSolution;
    int initCout=this.graphe.getCouts(sommets);
    List<List<Sommet>> voisins;
    List<Sommet> voisinAvecCoutMin;
    int coutVoisinMin;

    while (true) {
        voisins=super.generateVoisin(sommets,0);
        voisinAvecCoutMin=super.getVoisinAvecCoutMin(voisins);
        coutVoisinMin=this.graphe.getCouts(voisinAvecCoutMin);
        if (coutVoisinMin<initCout) {
            sommets=voisinAvecCoutMin;
            initCout=coutVoisinMin;
        } else {
            return sommets;
        }
    }
}
```

Exécution :

```
/****************Cout Descendente*****************/
Le temps d'execution descendente est 2 ns
cout optemiser descendente = 266
[3->, 4->, 2->, 5->, 9->, 0->, 7->, 1->, 8->, 6->]
```

2- Méthode de récuit simulé

- Choix de la solution initiale (voisin le plus proche), et une température T (T = n*10000)
- Tant que T>1 et le nombre d'itérations sans amélioration inférieur à n
- Choisir aléatoirement s' dans N(s)
- Générer un nombre réel aléatoire r dans [0,1]
- si r < exp(f(s)-f(s')/T) Alors [s=s']
- Mettre à jour T [T-100]

```
public class RecuitSimule extends AlgorithmeTrajectoir {

    private static double TMAX=10e6;


    public RecuitSimule(Graphe graphe) {

        super(graphe);

    }
```

La methode solve qui va nous retourner le chemain le plus court :

```
@Override

public List<Sommet> solve(List<Sommet> initialSolution) {

    List<Sommet> sommets=initialSolution;

    List<List<Sommet>> voisins;//list des voisins generer

    List<Sommet> voisinAvecCoutMin;//le voisin optimom

    double T=TMAX;//température initiale
```

```
    boolean conditionEquilibre=false;
    Random rand=new Random();
    while (T>1) {
        do {
            int j=rand.nextInt(graphe.getNbVille());
            voisins=generateVoisin(sommets, j);//géneration des
voisin

voisinAvecCoutMin=getVoisinAvecCoutMin(voisins);//choisir le
meilleur voisin
            var r=Math.random();//géneration de 0<r<1
            if (r<fonctionBoltZmann(voisinAvecCoutMin, sommets, T))
{
                sommets=voisinAvecCoutMin;
                conditionEquilibre=true;
            }
        } while (! conditionEquilibre);
        T=T-100;//Modification de la température
    }
    return sommets;
}
```

Execution :

```
/*****************Cout Recuit Simule******************/
Le temps d'execution est recuitSimule 484 ns
cout optemiser recuitSimule = 228
[0->, 4->, 2->, 1->, 8->, 5->, 9->, 6->, 7->, 3->]
```

3- Méthode tabou

Algorithme utilisé

- Choix de la solution initiale (voisin le plus proche), poser T = vide et s*=s

- Tant que le nombre d'itérations est inférieur à un nombre fixé
- Déterminer une solution s' qui minimise f(s') dans N(s)
- Si f(s') < f(s*) Alors [s*=s']
- Poser s=s' et mettre à jour TLa liste taboue qu'on a introduit regroupe les permutations récentes effectués a la place d'enregistrer les
- trajets, afin de minimiser l'utilisation en mémoire, surtout pour les problèmes de grande taille, on a fixé la
- mémoire de cette liste à n/2 (n = nombre de villes)

```
 public class Tabou extends AlgorithmeTrajectoir {
/**
 * constructeur
 * @param graphe
 */
public Tabou(Graphe graphe) {
    super(graphe);
}
```

La méthode solve dans la classe Tabou

```
@Override
public ArrayList<Sommet> solve(List<Sommet> initialSolution) {
    List<Sommet> sommets=initialSolution;//sommet
    int initCout=this.graphe.getCouts(sommets);//le cout initial
    List<List<Sommet>> voisins;//list des voisins generer
    List<Sommet> voisinAvecCoutMin;//le voisin optimom
    int coutVoisinMin;//le cout de voisin optimom
    List<List<Sommet>> tabou=new ArrayList<>();//le tableau Tabou
    int i=0;//nombre d'iteration
    int j=0;//l'indice de sommet pour la permutation

    while (i++<1000) {
        voisins=super.generateVoisin(sommets, j);//generation du
voisin de l'indice j
        voisinAvecCoutMin=super.getVoisinAvecCoutMin(voisins);//le
meilleur voisin
        coutVoisinMin=this.graphe.getCouts(voisinAvecCoutMin);//le
cout du meilleur voisin
        if (! tabou.contains(voisinAvecCoutMin)) {//test si la
solution etait deja pri
            tabou.add(sommets);
        }
        if (coutVoisinMin<initCout) {//changement du solution
initil
            sommets=voisinAvecCoutMin;
```

```
            initCout=coutVoisinMin;

            j=0;

        } else {

            if (j >= this.graphe.getNbVille()-1) {//test si
l'indice de permutation depasse la taille max

                j=0;

            }

            j++;

        }

    }

    return (ArrayList<Sommet>) sommets;

}
```

Exécution :

```
/****************Cout Tabou*****************/
Le temps d'execution tabou est 44 ns
cout optemiser tabou = 256
[3->, 4->, 7->, 5->, 9->, 0->, 2->, 1->, 8->, 6->]
```

4-Méthode VNS :

L algorithme VNS est base sur la méthode descente.et utilise des voisinages successifs pour atteindre un optimum local.

-Un ensemble de structures de voisinage Nk o`u (k = 1, ..., n) est défini.

-chaque itération de l algorithme est composée de trois étapes: secouage, recherche locale et déplacement.

-on utilise l operateur k−opt pour différentes valeurs de k(k = 2, 3, 4).

```java
public class VNS extends AlgorithmeTrajectoir {


    public VNS(Graphe graphe) {
        super(graphe);
    }
```

```java
     @Override
public List<Sommet> solve(List<Sommet> initialSolution) {
    List<Sommet> sommets=initialSolution;
    int initCout=this.graphe.getCouts(sommets);
    List<Sommet> voisinAvecCoutMin;
    List<Sommet> voisinAvecDes;


    int coutVoisinAvecDes;
    int k=0;
    int iter=0;
    while (iter++<1000000) {
        while (k<=4) {
            voisinAvecCoutMin=generationVoisin(k, sommets);
            voisinAvecDes=new
Descendente(graphe).solve(voisinAvecCoutMin);
            coutVoisinAvecDes=graphe.getCouts(voisinAvecDes);
            if (coutVoisinAvecDes<initCout) {
                sommets=voisinAvecDes;
                initCout=coutVoisinAvecDes;
```

```
                k=1;
            } else {
                k++;
            }
        }
    }
    return sommets;
}
```

Execution :

```
/****************Cout VNS******************/
Le temps d'execution tabou est 11ns
cout optemiser VNS = 262
[2->, 4->, 0->, 5->, 8->, 1->, 7->, 1->, 9->, 6->]
```

5 - Simulations

Matrice avec 20 sommets :

```
/***************Initial Solution******************/
le cout du solution initial 649
[1->, 13->, 3->, 17->, 19->, 0->, 15->, 4->, 7->, 14->, 12->, 6->, 9->, 2->, 18->, 5->, 8->, 10->, 16->, 11->]
/***************Cout Descendente*****************/
Le temps d'execution descendente est 5 ns
cout optemiser descendente = 405
[4->, 13->, 3->, 17->, 10->, 0->, 15->, 12->, 7->, 14->, 1->, 6->, 9->, 2->, 18->, 19->, 8->, 5->, 16->, 11->]
/***************Cout Tabou*****************/
Le temps d'execution tabou est 119 ns
cout optemiser tabou = 299
[16->, 17->, 3->, 13->, 10->, 0->, 15->, 12->, 7->, 14->, 1->, 6->, 9->, 18->, 2->, 4->, 8->, 5->, 19->, 11->]
/***************Cout Recuit Simule*****************/
Le temps d'execution est recuitSimule 1091 ns
cout optemiser recuitSimule = 291
[1->, 13->, 3->, 19->, 8->, 5->, 12->, 11->, 7->, 14->, 10->, 6->, 9->, 18->, 2->, 4->, 15->, 0->, 17->, 16->]
/***************Cout VNS*****************/
Le temps d'execution tabou est 13ns
cout optemiser VNS = 336
[19->, 13->, 3->, 16->, 17->, 0->, 15->, 5->, 7->, 14->, 1->, 6->, 9->, 2->, 18->, 10->, 8->, 4->, 12->, 11->]

Process finished with exit code 0
```

Matrice avec 30 sommets :

```
/***************Initial Solution******************/
le cout du solution initial 1206
[9->, 19->, 22->, 10->, 14->, 18->, 21->, 25->, 28->, 0->, 20->, 7->, 11->, 13->, 27->, 12->, 16->, 29->, 2->, 24->, 23->, 15->, 26->, 4->, 5->, 6->,
  8->, 17->, 3->, 1->]
/***************Cout Descendente*****************/
Le temps d'execution descendente est 12 ns
cout optemiser descendente = 593
[19->, 9->, 14->, 10->, 26->, 18->, 21->, 25->, 29->, 0->, 20->, 5->, 11->, 13->, 27->, 12->, 16->, 7->, 2->, 24->, 23->, 15->, 22->, 4->, 3->, 6->, 8->,
   17->, 28->, 1->]
/***************Cout Tabou*****************/
Le temps d'execution tabou est 154 ns
cout optemiser tabou = 404
[19->, 9->, 14->, 1->, 26->, 10->, 12->, 15->, 29->, 0->, 20->, 5->, 11->, 25->, 27->, 3->, 18->, 7->, 2->, 24->, 23->, 13->, 22->, 4->, 21->, 6->, 8->,
  17->, 28->, 16->]
/***************Cout Recuit Simule*****************/
Le temps d'execution est recuitSimule 2030 ns
cout optemiser recuitSimule = 325
[15->, 18->, 21->, 19->, 12->, 26->, 10->, 14->, 25->, 24->, 2->, 7->, 13->, 11->, 5->, 3->, 16->, 6->, 20->, 0->, 29->, 9->, 22->, 17->, 27->, 23->,
  8->, 28->, 1->, 4->]
/***************Cout VNS*****************/
Le temps d'execution tabou est 13ns
cout optemiser VNS = 500
[22->, 9->, 19->, 10->, 28->, 18->, 21->, 27->, 13->, 0->, 29->, 7->, 11->, 5->, 14->, 12->, 16->, 3->, 2->, 24->, 23->, 15->, 25->, 4->, 20->, 6->, 8->,
   17->, 26->, 1->]
```

Conclusion

Le problème du voyageur de commerce est toujours d'actualité dans la recherche en informatique, étant donné le nombre important de problèmes réels auxquels il correspond. Les problèmes dérivés et les extensions en sont très nombreux. Par exemple, des fenêtres de temps peuvent y être ajoutées. Ce concept consiste à imposer des contraintes de temps pour la traversée de chaque sommet. Autre exemple, il peut y avoir plusieurs voyageurs de commerce partant d'un même sommet, ou de sommets différents. Il suffit alors de considérer que les voyageurs de commerce sont des véhicules pour arriver à des problèmes de tournées de véhicules : étant donnée une flotte de véhicules, le problème consiste à déterminer les trajets de chacun pour livrer à moindre coût des clients en marchandise (chaque client est représenté par un sommet dans le graphe). Le nombre de véhicules peut être fixe ou non, les capacités des véhicules peuvent être les mêmes ou non, des fenêtres de temps peuvent être définies... Pour chacune de ces variantes, de nouvelles méthodes peuvent être explorées.