Blin Sébastien
Collin Pierre-Henri

ECOLE SUPERIEURE
D'INGENIEURS DE RENNES

École supérieure d'ingénieurs de Rennes

1ère Année
Parcours Informatique

Algorithmie et complexité

Compte-Rendu TP4

Sous l'encadrement de :

Ridoux Olivier
Maurel Pierre

1 Algorithme génétique : KS

1.1 Objectif

1.2 Moyens mis en œuvre

1.3 Résultats

2 Conclusion

3 Algorithme génétique : TSP

3.1 Objectif

Dans la seconde partie de ce dernier TP, nous avons utilisé l'algorithme génétique précédement implémenté (en ne modifiant que la partie client et la représentation interne de l'individu) afin de résoudre un problème assez différent, le problème du voyageur de commerce.
Dans ce problème, les données nous fournissaient des coordonnées de villes (x,y). Le but était alors de trouver le chemin le plus court passant par tous les points décrit par les données fournies.

3.2 Moyens mis en œuvre

Pour résoudre ce problème, nous avons gardé la classe Population utilisée dans le problème KS, même si d'autres méthodes existent (`http://endomorphis.me/blog/2013/10/11/modele-de-parallelisme-en-ile` `http://en.wikipedia.org/wiki/Genetic_algorithm`, etc.). En effet, la méthode de sélection ainsi que de reproduction fonctionne exactement de la même manière pour le problème du voyageur de commerce et pour le problème du sac à dos.
Pour ce qui s'agit de la partie client, nous avons repris l'algorithme du précédent client, que nous avons un peu modifié pour qu'il fonctionne avec nos individus.

```
/* parametres */
int nbr_indiv=100;
double prob_mut=0.008;
int nbr_villes = 64;
double[] coord_x = new double[nbr_villes];
double[] coord_y = new double[nbr_villes];
charge_coords("../data_vdc/"+nbr_villes+"coords.txt",nbr_villes, coord_x, coord_y);
Individu_VDC[] pop = new Individu_VDC[nbr_indiv];
for(int i = 0; i < nbr_indiv; ++i)
        pop[i] = new Individu_VDC(coord_x, coord_y);
Population _p = new Population(pop);

/* on genere les generations successives
 * en faisant se reproduire la population
 * et on affiche l'adaptation moyenne et maximale de chaque generation
 * on s'arrete si on a atteint la capacite ou si on fait un nombre donne (parametre) d'iterations
 * le resultat est alors donne par l'individu maximal de la derniere generation
 */
int it = 0;
double best = -1;
```

```
Display_VDC disp = new Display_VDC((Individu_VDC)_p.individu_maximal());
while(it < 1000)
{
        System.out.println("generation " + it);
        _p.reproduction(prob_mut);
        best = 1/((Individu_VDC)_p.individu_maximal()).adaptation();
        System.out.println("Score : " + best);
        if(it%50 == 0)
                disp.refresh((Individu_VDC)_p.individu_maximal());
        ++it;
}
disp.refresh((Individu_VDC)_p.individu_maximal());
int[] val= ((Individu_VDC)_p.individu_maximal()).get_parcours();
System.out.println("Resultat : ");
for(int b : val)
        System.out.print(b);
System.out.println("\nAvec un score de :"+_p.individu_maximal().adaptation());
```

Pour la représentation interne, un tableau d'entier nous paraissait le plus facile à manipuler. L'individu contient donc un tableau d'entier représentant l'ordre dans lequel le voyageur visite les villes. À la création, le tableau et intialisé dans un ordre aléatoire avec des numéros entre 0 et le nombre de villes -1.

```
private double[] _x;
private double[] _y;
private int[] _ind;

//Constructeur
public Individu_VDC(double[] coord_x, double[] coord_y) {
        _x = coord_x;
        _y = coord_y;
        _ind = new int[_x.length];
        for(int i = 0; i < _ind.length; ++i)
                _ind[i] = i;
        shuffleArray(_ind);
}

public static void shuffleArray(int[] a) {
        int n = a.length;
        Random random = new Random();
        random.nextInt();
        for (int i = 0; i < n; i++) {
                int change = i + random.nextInt(n - i);
                swap(a, i, change);
        }
}
```

L'adaptation consiste à calculer la longueur du trajet. Dans ce cas, plus le score est minimal, plus la réponse est bonne. Et comme miniser revient à maximiser son inverse, on considère 1/la longueur du trajet.

```
@Override
public double adaptation() {
        double adapt = 0;
        for(int i = 1; i < _ind.length; ++i)
                adapt += Math.abs(Math.sqrt(Math.pow(_x[_ind[i-1]]-_x[_ind[i]],2)+Math.pow(_y[_in
        return 1/adapt;
}
```

Pour le croisement, nous avions le choix entre plusieurs algorithmes. Nous avons hésité entre le croisement ordinal et celui proposé sur `http://labo.algo.free.fr/pvc/algorithme_genetique.html`. Nous avons implémenté le second car nous ne l'avions pas encore vu en cours.

```
@Override
public Individu[] croisement(Individu conjoint) {
        Individu[] res = new Individu[1];
        Individu_VDC ind = new Individu_VDC(_x,_y);
        int index = 0;
        Random random = new Random();
        index = random.nextInt(_ind.length-1);
        //On copie la premiere partie du parent 1
        for(int i=0; i<_ind.length; ++i)
                ind._ind[i] = -1;
        for(int i=0; i<=index; ++i)
                ind._ind[i] =_ind[i];
        ++index;
        for(int i=0; i<_ind.length; ++i)
        {
                boolean p = false;
                for(int t : ind._ind)
                        if(t == ((Individu_VDC)conjoint)._ind[i])
                                p = true;
                if(!p)
                {
                        ind._ind[index]=((Individu_VDC)conjoint)._ind[i];
                        ++index;
                }
        }
        res[0] = ind;
        return res;
}
```

Enfin pour la partie mutation, nous avons décidé qu'elle serait modélisée par un échange de 2 villes.

```
@Override
public void mutation(double prob) {
        for(int i = 0; i < _ind.length; ++i)
                if(Math.random() < prob)
                {
                        int r = (int)(Math.random()*((double)_ind.length));
                        int temp = _ind[r];
```

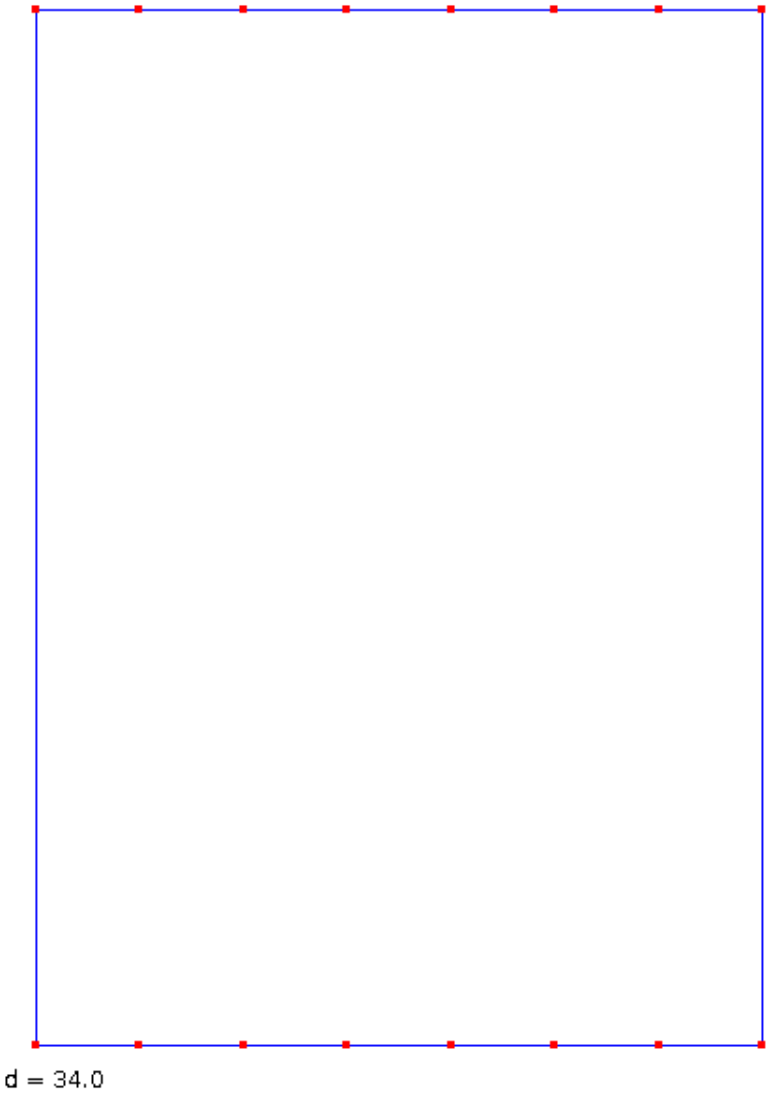
```
                _ind[r] = _ind[i];
                        _ind[i] = temp;
                }
}
```

3.3 Résultats

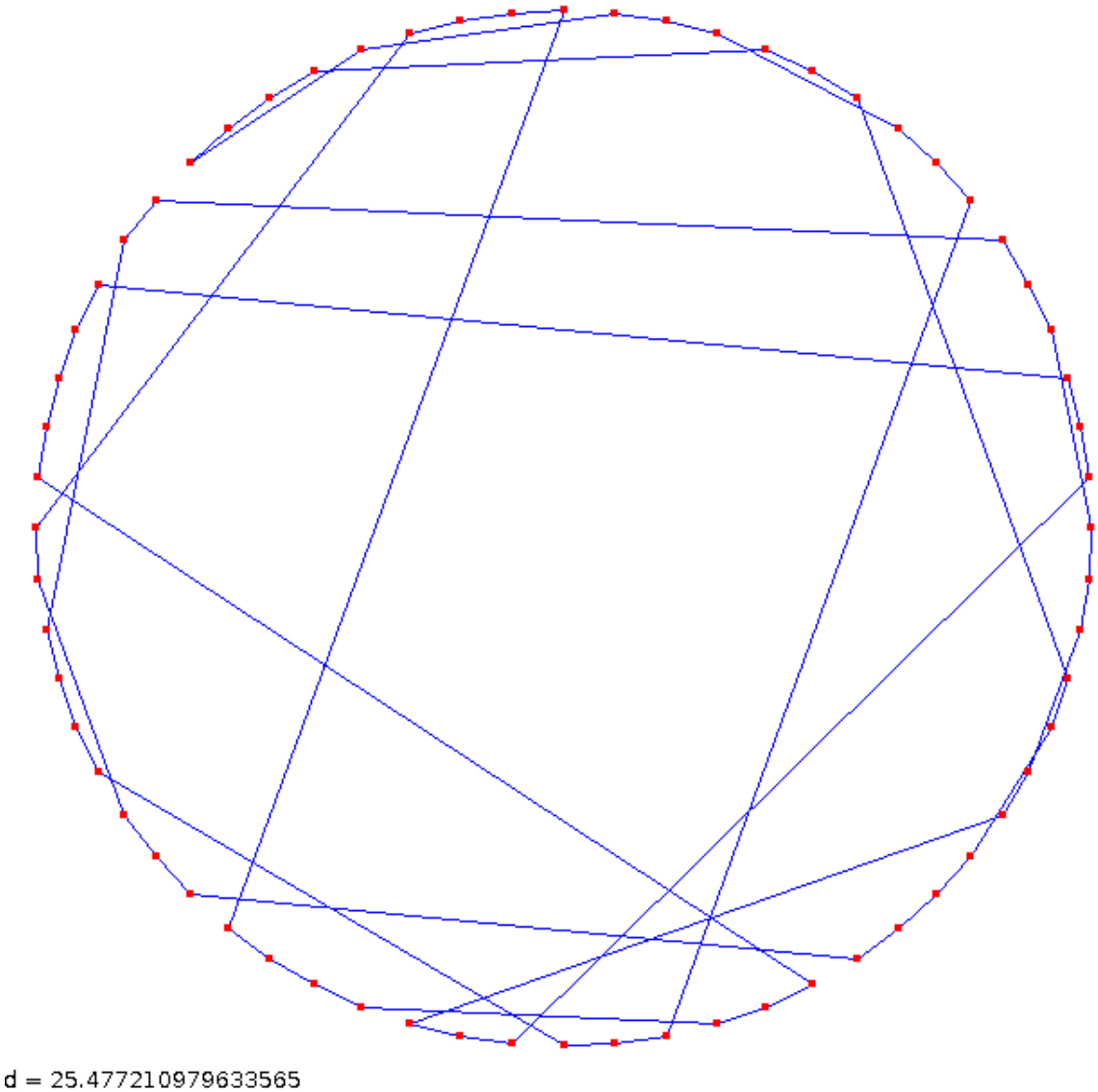
Nous avons alors testé notre algorithme sur des données pour 4, 16, 64 et 250 villes. On remarque que l'algorithme génétique donne rapidement de bons résultats (voir le meilleur résultat) pour un nombre assez faible de ville. À partir de 64 villes, il devient difficile de trouver de meilleurs résultats, l'algorithme commence à converger vers une solution différente de la meilleure solution. Nous remarquons qu'il faut alors passer par d'autres méthodes. Il est possible de choisir d'autres méthodes de reproduction par exemple ou encore mieux, trouver une optimisation ! En effet, à la fin du cours, nous avons vu qu'avec une petite optimisation (que nous n'avons pas implémenté) de commencer avec un score très proche de 12 dès le début de l'algorithme. (11 étant la meilleure solution).
On peut voir les résultats de notre algorithme avec les figures suivantes.

4 Conclusion

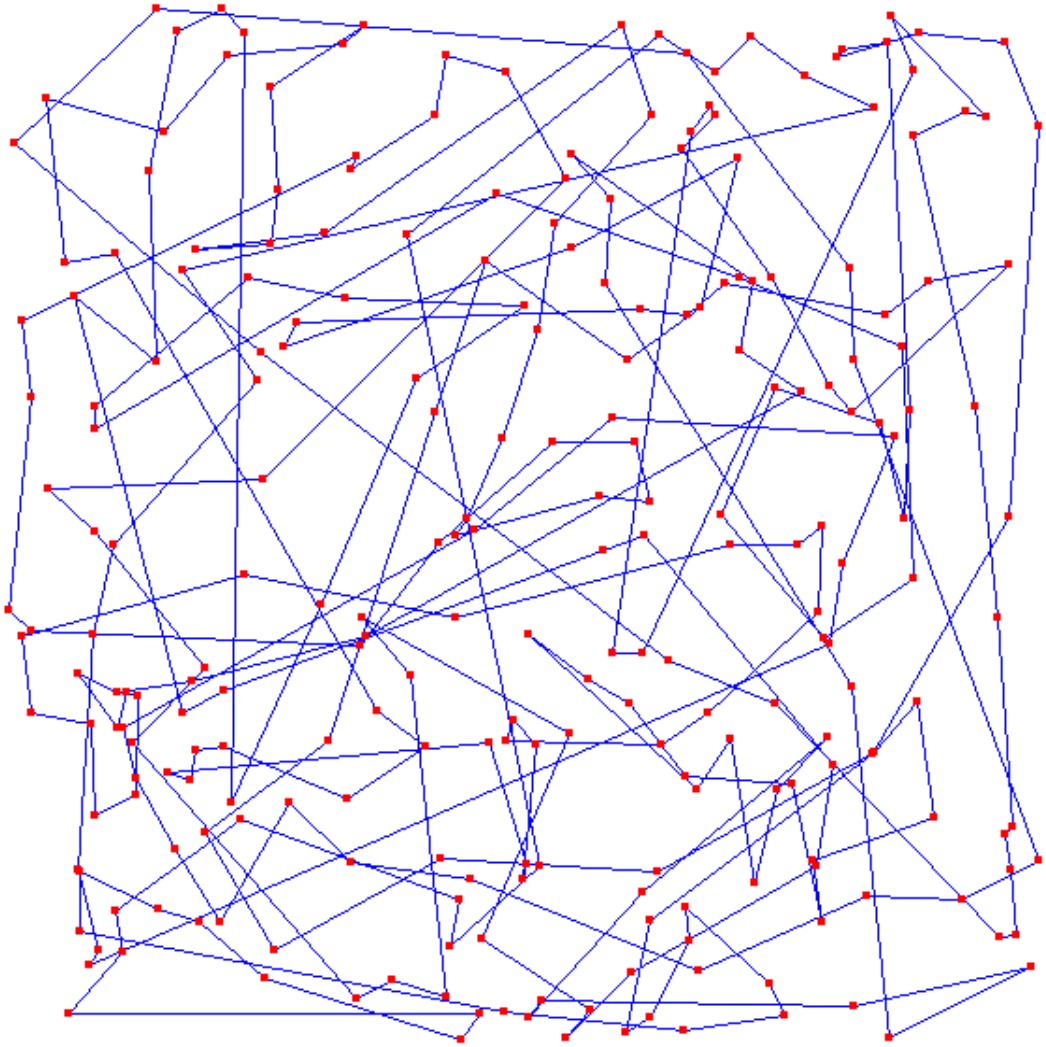
On peut donc voir que l'algorithmie génétique est un outil très puissant qui permet de trouver rapidement des solutions convenables à des problèmes compliqués comme KS, TSP, etc. Mais généralement, une optimisation est nécessaire pour tirer pleinement profit de ce type de méthode. En effet, dans le cas de TSP, pour un nombre important de villes, l'algorithme a du mal à trouver une bonne solution lorsqu'il part de rien. Il nécessite alors des optimisations pour pouvoir converger vers une meilleure solution.



Solution pour 16 villes, après 32 générations



Solution pour 64 villes, après 1000 générations



Solution pour 250 villes, après 1000 générations