MASTER I INFORMATIQUE 2005/2006

LIU YINAN

Rapport de Mémoire

Application des Algorithmes Génétiques au Problème du Voyageur de Commerce

Encadrants :
M. Nakechbandi
J. Boukachour

UNIVERSITÉ DU HAVRE

Table des matières

Résumé

Le problème du voyageur de commerce étant combinatoire (le nombre de solutions est de l'ordre n !, n nombre de villes), le but du projet est d'appliquer des algorithmes génétiques pour trouver le meilleur circuit de coût minimal. La mise en oeuvre de cet algorithme consiste particulièrement à intégrer des nouveaux opérateurs de sélection et de croisement génétique.

Remerciements

Je tiens à remercier :

M. Nakechbandi et M. Boukachour pour leur encadrement et leur nombreuses explications du sujet.

Chapitre 1

Introduction

Les algorithmes génétiques ont été inventés par Jonh Holland dans les années 60. Repris notamment par Golberg dans les années 70,Le principe des algorithmes génétiques s'inspire directement des lois de la sélection naturelle, décrites par Darwin. cette technique connait aujourd'hui un franc succès. On l'utilise dans la résolution de problèmes complexes, nécessitant des temps de calcul élevés.

Les algorithmes génétiques sont des algorithmes d'optimisation s'appuyant sur des techniques dérivées de la génétique et des mécanismes d'évolution de la nature : croisements, mutations, sélections, etc... Ils appartiennent à la classe des algorithmes évolutionnaires.

Les applications des AG sont multiples : traitement d'image (alignement de photos satellites, reconnaissance de suspects...), optimisation d'emplois du temps, optimisation de design, apprentissage des réseaux de neurones [Renders, 1995], etc.

Le but de ce projet est d'appliquer l'algorithme génétique au problème de voyageur de commerce, la mise en œuvre de cet algorithme consiste particulièrement à intégrer des nouveaux opérateurs de sélection et de croisement génétique.

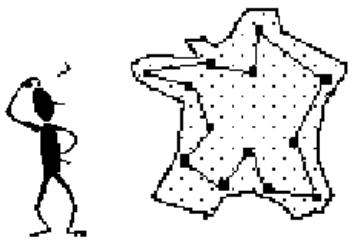
Chapitre 2

Problème de Voyageur de Commerce

2.1 Théoriquement

soit un graphe G = (S, A) , problème est de trouver une parcours qui passe par tous les sommets une et une seule fois et qui soit de poids minimal. En general, on travaille sur un graphe complet.

2.2 Pratiquement



Un voyageur de commerce doit visiter *N* villes données en passant par chaque ville exactement une fois. Il commence par une ville quelconque et termine en retournant à la ville de départ. Les distances entre les villes sont connues. Quel chemin faut-il choisir afin de minimiser la distance parcourue ? La notion de distance

peut-être remplacée par d'autres notions comme le temps qu'il met ou l'argent qu'il dépense : dans tous les cas, on parle de coût.

2.3 Complexité du Problème

La complexité du problème croît comme la factorielle du nombre de villes. Si l'on considère des permutations simples, il existe 3 628 000 permutations de 10 villes. Pour 100 villes, on passe à 10 puissance 158. La complexité du problème croît de manière plus que polynomiale (problème dit NP).

n	***Nombre de possibilités***	***Temps de calcul***
5	12	micro-seconde
10	181440	deuxième de seconde
15	43 milliards	dizaine d'heures
20	60.10^{15}	milliers d'années
25	310.10^{21}	milliards d'années

2.4 Intérêt

Le PVC fournit un exemple d'étude d'un problème NP-complets dont les méthodes de résolution peuvent s'appliquer à d'autres problèmes de mathématiques discrète. Néanmoins, il a aussi des applications directes, notamment dans les transports et la logistique. Par exemple, trouver le chemin le plus court pour les bus de ramassage scolaire ou, dans l'industrie, pour trouver la plus courte distance que devra parcourir le bras mécanique d'une machine pour percer les trous d'un circuit imprimé (les trous représentent les villes).

Chapitre 3

Algorithme Génétique

Les algorithmes génétiques s'inspirent de l'évolution des espèces dans leur cadre naturel. Les espèces s'adaptent à leur cadre de vie qui peut évoluer, les individus de chaque espèce se reproduisent, créant ainsi de nouveaux individus, certains subissent des modifications de leur ADN, certains disparaissent ....

Un algorithme génétique va reproduire ce modèle d'évolution dans le but de trouver des solutions pour un problème donné.

1. Une population sera un ensemble d'individus.
2. Un individu sera une réponse à un problème donné, qu'elle soit ou non une solution valide du problème.
3. Un gène sera une partie d'une réponse au problème, donc d'un individu.
4. Une génération est une itération de notre algorithme.

Un algorithme génétique va faire évoluer une population dans le but d'améliorer les individus.

Le déroulement d'un algorithme génétique peut être découpé en cinq parties :

1. La création de la population initiale.
2. L'évaluation des individus.
3. La création de nouveaux individus.
4. L'insertion des nouveaux individus dans la population.
5. Réitération du processus.

3.1 La Création de la Population Initiale

La population initiale sera créée de manière aléatoire à condition que chaque individus dans la populations créée soit une solution du problème. La taille de la population initiale doit être raisonnablement grande en tenant compte à la fois de la qualité des solutions trouvées et du temps d'exécution de notre algorithme.

3.2 L'Évaluation des Individus

Après avoir créé la population initiale, nous attribuons une valeur d'adaptation ou une 'note' à chaque individu selon leur performance par rapport le coût de distance totale. Il faudrait donc créer une fonction d'évalution pour évaluer la qualité de chaque individu.

3.3 La Création de Nouveaux Individus

Une fois que nous avons fini d'évaluer nos individus, nous allons créer de nouveaux individus qui seront la nouvelle génération de notre population, et nous esperons que certain entre eux seront de meilleurs solutions à notre problème.

Dans notre programme, nous allons utiliser des différents opérateurs pour céer notre nouvelle génération.

3.3.1 Les Selections

3.3.1.1 La Selection par Roulette

Les parents sont sélectionnés en fonction de leur performance. Meilleur est le résultat codé par un chromosome, plus grandes sont ses chances d'être sélectionné. Il faut imaginer une sorte de roulette de casino sur laquelle sont placés tous les chromosomes de la population, la place accordée à chacun des chromosomes étant en relation avec sa valeur d'adaptation.

Ensuite, la bille est lancée et s'arrête sur un chromosome. Les meilleurs chromosomes peuvent ainsi être tirés plusieurs fois et les plus mauvais ne jamais être sélectionnés. Cela peut être simulé par l'algorithme suivant :

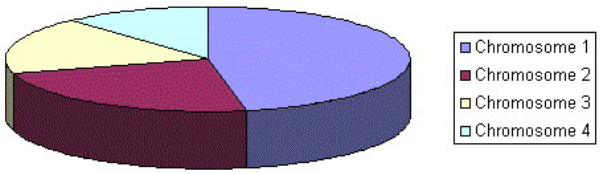


FIG. 3.1 – Schéma d'une roulette

1. On calcule la somme ***S1*** de toutes les fonctions d'évaluation d'une population.
2. On génère un nombre ***r*** entre 0 et ***S1***.
3. On calcule ensuite une somme ***S2*** des évaluations en s'arrêtant dès que ***r*** est dépassé.
4. Le dernier chromosome dont la fonction d'évaluation vient d'être ajoutée est sélectionné.

3.3.1.2 La Selection par Rang

La sélection précédente rencontre des problèmes lorsque la valeur d'adaptation des chromosomes varient énormément. Si la meilleure fonction d'évaluation d'un chromosome représente 90% de la roulette alors les autres chromosomes auront très peu de chance d'être sélectionnés et on arriverait à une stagnation de l'évolution.

La sélection par rang trie d'abord la population par leur scores. Ensuite, chaque chromosome se voit associé un rang en fonction de sa position. Ainsi le plus mauvais chromosome aura le rang 1, le suivant 2, et ainsi de suite jusqu'au meilleur chromosome qui aura le rang N. La sélection par rang d'un chromosome est la même que par roulette, mais les proportions sont en relation avec le rang plutôt qu'avec la valeur de l'évaluation. Le tableau suivant fournit un exemple de sélection par rang. Avec cette méthode de sélection, tous les chromosomes ont une chance d'être sélectionnés. Cependant, elle conduit à une convergence plus lente vers la bonne solution. Ceci est dû au fait que les meilleurs chromosomes ne diffèrent pas énormément des plus mauvais.

Chromosomes	***1***	***2***	***3***	***4***	***5***	***6***	**Total**
Probabilités initiales	89%	5%	1%	4%	3%	2%	100%
Rang	6	5	1	4	3	2	21
Probabilités finales	29%	24%	5%	19%	14%	9%	9%

TAB. 3.1 – Exemple de sélection par rang pour 6 chromosomes.

3.3.1.3 La Selection par Tournoi

Sur une population de ***m*** chromosomes, on forme ***m*** paires de chromosomes. Dans les paramètres de l'AG, on détermine une probabilité de victoire du plus fort. Cette probabilité représente la chance qu'a le meilleur chromosome de chaque paire d'être sélectionné. Cette probabilité doit être grande (entre 70% et 100%). A partir des ***m*** paires, on détermine ainsi ***m*** individus pour la reproduction.

3.3.1.4 L'Élitisme

A la création d'une nouvelle population, il y a de grandes chances que les meilleurs chromosomes soient perdus après les opérations d'hybridation et de mutation. Pour éviter cela, on utilise la méthode d'élitisme. Elle consiste à copier un ou plusieurs des meilleurs chromosomes dans la nouvelle génération. Ensuite, on génère le reste de la population selon l'algorithme de reproduction usuel. Cette méthode améliore considérablement les algorithmes génétiques, car elle permet de ne pas perdre les meilleurs solutions.

3.3.2 Les Croisements

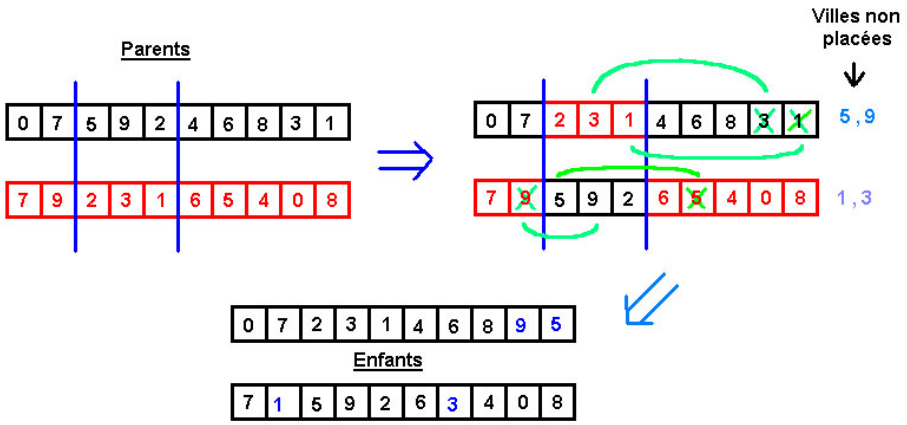
3.3.2.1 Le Croisement CPA

C'est une méthode qui s'agit un croisement en deux points.

Nous suivons la méthode suivante :

1. On choisi aléatoirement deux points de découpe.
2. On interverti, entre les deux parcours, les parties qui se trouvent entre ces deux points.
3. On supprime, à l'extérieur des points de coupe, les villes qui sont déjà placées entre les points de coupe.

4. On recense les villes qui n'apparaissent pas dans chacun des deux parcours.
5. On remplit aléatoirement les trous dans chaque parcours.



3.3.2.2 Le Croisement 1X

Il est désigne par le sigle 1X(ou 1 rappelle qu'il s'agit d'un croisement en 1 point, X symbolise les croisements ; en toute rigueur, il faudrait le designer par 1OX pour croisement d'ordre a 1 point). Il commence comme l'opérateur simple a un point par une recopie du début du pere(resp. de la mere) et il complète la fin du chromosome avec les valeurs numériques manquantes dans l'ordre de la mere (resp. du pere). En echangeant «debut» et «fin» dans la description de ce croisement, on peut obtenir un autre fils et une autre fille selon un principe similaire ; ceci peut permettre, par exemple, de garder les deux meilleurs enfants obtenus, ce qui améliore la qualité de ce croisement. le tableau illustre les étapes du croisement 1X.

$$p1 = (123|456789) \qquad p2 = (452|187693)$$

$$e1 = (123|8xxxxx) \qquad e2 = (456|7xxxxx)$$

$$e1 = (123|87xxxx) \qquad e2 = (456|78xxxx)$$

$$e1 = (123|876xxx) \qquad e2 = (456|789xxx)$$

$$e1 = (123|8769xx) \qquad e2 = (456|7891xx)$$

$$e1 = (123|87694x) \qquad e2 = (456|78912x)$$

$$e1 = (123|876945) \qquad e2 = (456|789123)$$

3.3.2.3 Le Croisement MPX

Le PMX fut proposé par Goldberg et Lingle [GL85]. Le but était de construire un enfant par choix de sous séquences d'un ordonnancement d'un des parents et de préserver l'ordre et la position d'autant de lots que possible des autres parents. La sous séquence de l'ordonnancement est sélectionnée par choix de deux points de coupure aléatoire, lesquels servent de frontière pour l'opération de remplacement. Considérons par exemple les deux parents :

$$p1 = (123|4567|89) \qquad p2 = (452|1876|93)$$

Ces deux parents vont produire deux enfants. Premièrement, les segments compris entre les points de coupures sont échangés :

$$e1 = (xxx|1876|xx) \qquad e2 = (xxx|4567|xx)$$

L'échange défénit ainsi une série de permutations pour les autres lots : $1 \leftrightarrow 4$, $8 \leftrightarrow 5$, $7 \leftrightarrow 6$, $6 \leftrightarrow 7$.

Certaines positions de lots sont donc inchangées.

$$e1 = (x23|1876|x9) \qquad e2 = (xx2|4567|93)$$

Les deux enfants obtenus sont donc toujours viables (pas deux fois le même lot dans le vecteur) et on évite ainsi l'utilisation d'un opérateur de réparation.

Finalement, le premier x du vecteur enfant e1 est remplacé par 4 en raison de la permutation $1 \leftrightarrow 4$. De la même manière, on réalise les permutations pour les autres x. Les vecteurs enfants finaux sont donc :

$$e1 = (423|1876|59) \qquad e2 = (182|4567|93)$$

Comme nous venons de le voir, cet algorithme présente l'avantage de fournir des ordonnancements viables. Or, dans cet exemple, nous considérons uniquement des lots positifs (placement en marche avant). Nous avons donc modifié l'algorithme standard pour prendre en compte les deux sens de placement des lots : le placement en marche arrière et le placement en marche avant.

3.3.2.4 Le Croisement OX

Le crossover OX, proposé par Davis [Dav85a], construit un enfant par choix de sous-séquences d'un ordonnancement d'un des parents et préserve l'ordre relatif des lots des autres parents. Par exemple, soit deux parents

$$p1 = (123|4567|89) \qquad p2 = (452|1876|93)$$

Ces deux parents vont produire deux enfants. Premièrement, les segments entre points de coupures sont copiés.

$$e1 = (xxx|1876|xx) \qquad e2 = (xxx|4567|xx)$$

Ensuite, il suffit de partir du point de coupure d'un des parents, les lots de l'autre parent sont copiés dans le même ordre, omettant les symboles déjà présents. Lorsque l'on atteint la fin du vecteur, on continue depuis le premier emplacement de la chaîne. La séquence de lots issue du second parent est donc :

$$9-3-2-1-8$$

Cette séquence est placée dans le premier enfant. Le point de départ est la seconde coupure.

$$e1 = (218|4567|93)$$

De la même manière, on obtient le deuxième enfant :

$$e2 = (345|1876|92)$$

Dans ce cas aussi, les enfants sont toujours viables.

3.3.2.5 Le Croisement CX

Le crossover CX, proposé par Oliver [OSH87], engendre un enfant dans tous les cas tel que chaque lot provient d'un des parents.

$$p1 = (123456789) \qquad p2 = (412876935)$$

L'enfant engendré est obtenu en prenant le premier lot du premier parent

$$e1 = (1xxxxxxxx) \qquad e2 = (4xxxxxxxx)$$

Ensuite, chaque lot doit être pris dans un des parents, à la même position. Il n'y a pas de choix possible. Le prochain lot à considérer est le lot 4.

$$e1 = (1xx4xxxxx) \qquad e2 = (4xx8xxxxx)$$

Le lot 4, dans le parent $p1$, implique le lot 8 dans le parent $p2$.

$$e1 = (1xx4xxx8x) \qquad e2 = (4xx8xxx3x)$$

On continue ainsi jusqu'à la fin du cycle et on obtient :

$$e1 = (1234xxx8x) \qquad e2 = (4128xxx3x)$$

On complète alors en échangeant les lots restants des deux parents, on obtient :

$$e1 = (123476985) \qquad e2 = (412856739)$$

3.3.3 La Mutation

La mutation génère des ' erreurs ' de recopie, afin de créer un nouvel individu qui n'existait pas auparavant. Le but est d'éviter à l'AG de converger vers des extrema locaux de la fonction et de permettre de créer des éléments originaux. Si elle génère un individu plus faible l'individu est éliminé. La probabilité de mutation représente la fréquence à laquelle les gènes d'un chromosome sont mutés.

– S'il n'y a pas de mutation, le fils est inséré dans la nouvelle population sans changement.
– Si la mutation est appliquée, une partie du chromosome est changée.

La mutation est prévue pour éviter au AG de s'enliser dans des optima locaux. Mais si elle est trop fréquente, le AG est orienté vers une recherche aléatoire de la bonne solution.

Généralement, La méthode de mutation consiste en une permutation de deux villes. Nous sommes certains que les individus mutés auront toujours la forme d'une solution potentielle car nous ne changeons que l'ordre des ville. Par exemple : {A,B,***C***,D,E,F,G,***H***,I,J} pourra être muté en {A,B,***H***,D,E,F,G,***C***,I,J}.

3.3.4 L'Insertion des Nouveaux Individus dans la Population

Une fois que nous avons créé de nouveaux individus que ce soit par croisement ou par mutation, il nous faut sélectionner ceux qui vont continuer à participer Íd' l'amélioration de notre population.

Une méthode relativement efficace consiste à insérer les nouveaux individus dans la population, à trier cette population selon le score(la valeur d'adaptation) de ses membres, et à ne conserver que les N meilleurs individus. Ici, dans notre programme, nous allons considérer que N soit la taille de la population.

3.3.5 Réitération du processus

Au fait, nous ferons les quatre étapes présédentes en incrementant le nombre de génération, tant que le nombre de génération est inférieur au nombre maximum de génération. Et le nombre maximum de génération peut être fixé avec un nombre assez grand selon le problème à résoudre.

Une fois le nombre maximum de générations atteint, vous obtenez une population de solutions. Mais rien ne vous dit que la solution théorique optimale aura été trouvée. Les solutions se rapprochent des bonnes solutions, mais sans plus. Ce n'est pas une méthode exacte.

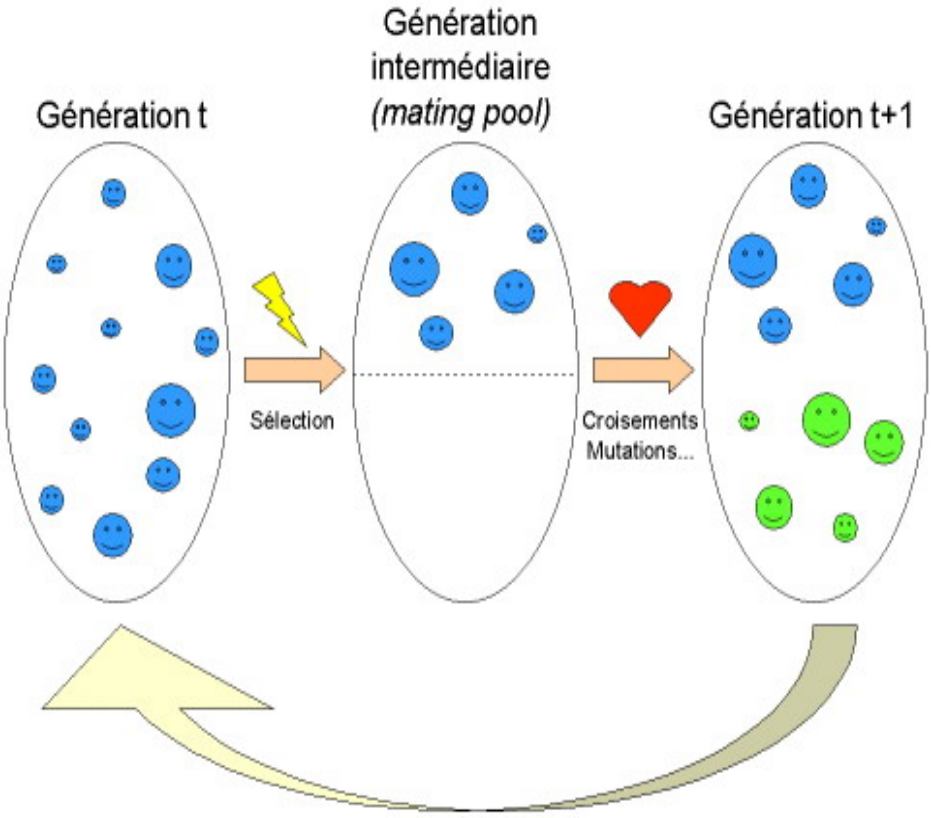


FIG. 3.2 – Schéma du fonctionnement de l'algorithme génétique

Chapitre 4

Application de l'AG au PVC

4.1 L'Objectif

L'objectif de ce projet est d'intégrer l'algorithme génétique avec les nouveaux opérateurs de selection et de croisement dans un ensemble des algorithmes qui peuvent tous être utilis'es à résoudre le problème de voyageur de commerce sous une interface graphique dans laquelle on pourra paramètriser et lancer l'algorithme génétique.

4.2 Les Statistics sur les Paramètres de L'AG

Comme il y a plusieurs types de l'opérapteur, et des paramètres, nous préférons bien savoir quelle est la différences de l'un et l'autre d'opérateur, et des choix de paramètre pour un résultat obtenu. J'ai donc réalisé une statistique qui s'est basé sur le même problème(le même plan des villes) et s'est stabilisé sur vight fois de tests en utilisant exactement les même paramètres et les même opérateurs.

Afin de voir la performance et l'éfficacité de chaque algorithme, j'ai également fait une statistique avec la distance finale et le temps d'exécution sur le même problème.

TAB. 4.1 – Tableau de Statistique sur les Paramètres de l'AG

Nombre de Villes	Croisement	Probabilité de Croisement	Selection	Probabilité de Mutation	Distance Moyenne	Ecart Type
20	CPA	***100%***	Roulette	0.1	121.4	2253
20	CPA	***60%***	Roulette	0.1	99.5	2136
20	CPA	100%	Roulette	***0.02***	151.0	2248
20	CPA	100%	Roulette	***0.2***	102.2	2244
20	***1X***	100%	Roulette	0.1	101.9	2159
20	***OX***	100%	Roulette	0.1	36.7	2068
20	CPA	100%	***Rang***	0.1	133.0	2289
20	CPA	100%	***Tounoi***	0.1	193.2	2322

TAB. 4.2 – Application des Algorithmes différents

Algorithmes	20 Villes		50 Villes		100 Villes	
	Distance	Temps(ms)	Distance	Temps(ms)	Distance	Temps(ms)
PPV	2371	0	3475	0	4607	0
Insertion	2025	0	3512	0	4695	10
2 Échange	1978	0	3107	0	4098	10
Prim	2326	11	3678	70	5288	531
l'Élastique	1975	3054	3073	45015	4652	580735
Recuit	1978	2056	3054	140	4213	541
Algo D	4315	0	8156	20	11916	40
AG	1941	1232	3013	3956	5913	17696

Chapitre 5

Conclusion

L'intérêt de ce projet a été pour moi, la mise en œuvre de l'algorithme génétique qui est largement utilisé dans la recherche scientifique.

Après avoir plusieurs tests du programme, j'ai découvert deux points où je me suis trompé dessus :

1. Premièrement, j'avait pensé que plus le nombre de population est grand, plus on a la chance de trouver la meillieur solution, cependant cette idée n'est pas correcte.

   En effet, surtout pour le grand nombre de villes, par exemple 200 villes, comme le nombre de population est très grand, le temps d'exécution est déjà archilong, et en même temps, la distance ne baisse pas rapidement non plus du fait que on tombe dans une population très large, et on n'a pas de grande chance de sélectionner les meilleur parents afin de produire la nouvelle génération. Par conséquent, quand le nombre maximum de génération atteint, on ne trouve pas une bonne solution que l'on a prévue.
2. En fait, le deuxième point est qu'il me reste, dans le programme, un problème que je n'arrive pas à rśoudre, c'est que lorsque le nombre de villes est passé 200, le résultat de l'AG est à peu près deux fois plus grand que celui de l'algorithme Plus Proche Voisin.

   Au départ, j'avais pensé que le nombre maximum de génération n'était pas assez grand, puis j'ai mis l'infinie pour le nombre maximum de génération, et j'ai également incrémenté le nombre de stagnation, la solution(distance) a été améliorée 45%(par exemple solution ancienne : 18560, solution actuelle : 10506), mais en tout cas, la solution n'est jamais inférieur à 10000, du coup la solution de plus proche voisin est 6750.

Chapitre 6

Références

6.1 Site Web

- Algorithme génétique : http://sis.univ-tln.fr/~tollari/TER/AlgoGen1/node5.html
- Chiffrement évolutionniste : http://www.revue-eti.net/document.php?id=858
- Résolution de problème Voyageur de commerce : http://wwwsi.supelec.fr/yb/projets/algogen/VoydeCom/VoyDeCom.html
- L'Algorithme génétique : http://www.chez.com/produ/badro/
- LesAlgorithmes génétiques : http://khayyam.developpez.com/articles/algo/genetic/#L3.1.3

6.2 Documents

- Thèse de Doctorat de l'Université du Havre : *TheseJeanPhilippeVacher.pdf (367 pages)*
- Algorihthmes de Graphe : *chap3_lacomme.pdf (27 pages)*

Annexe A

L'Interface Graphiique

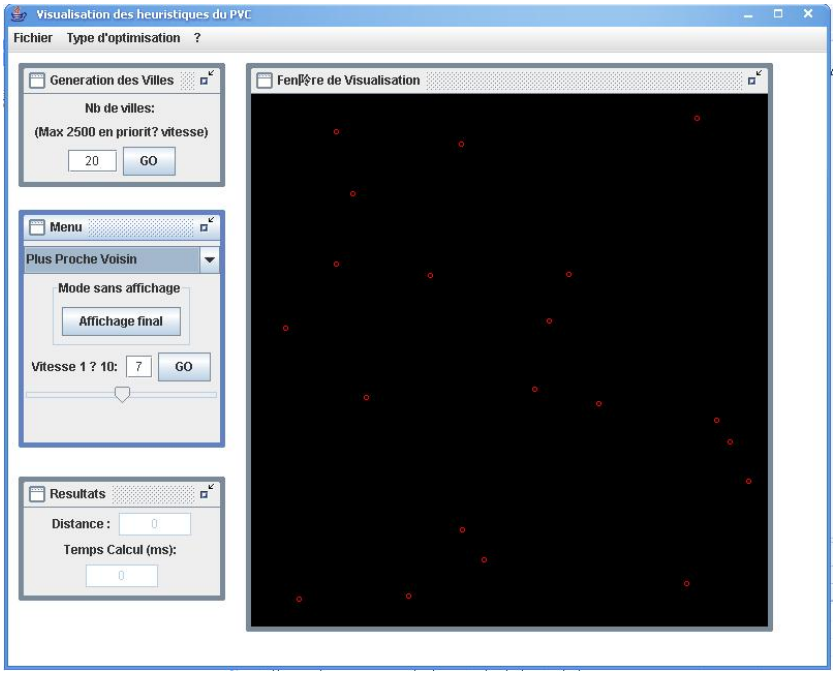


FIG. A.1 – L'interface avant le lancement des algorithmes

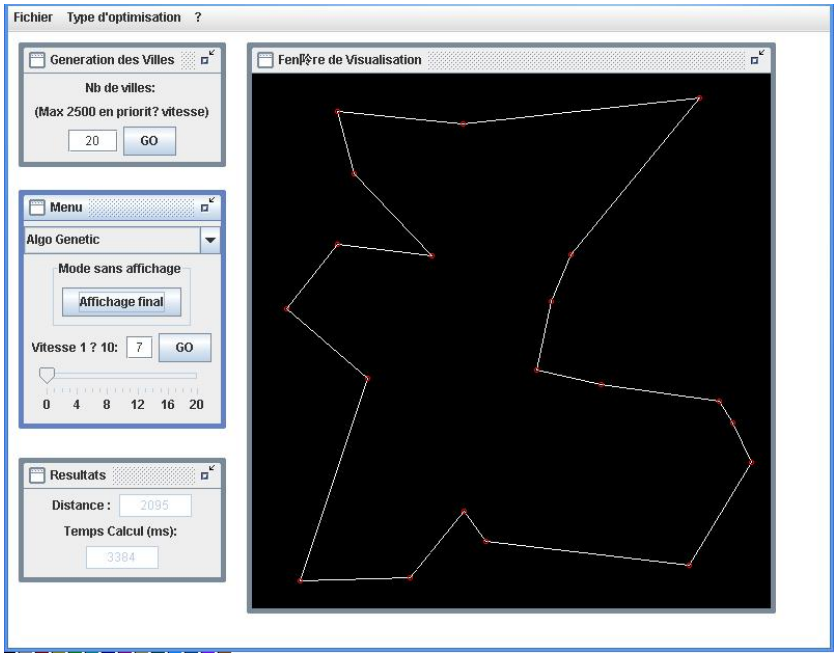


FIG. A.2 – L'interface

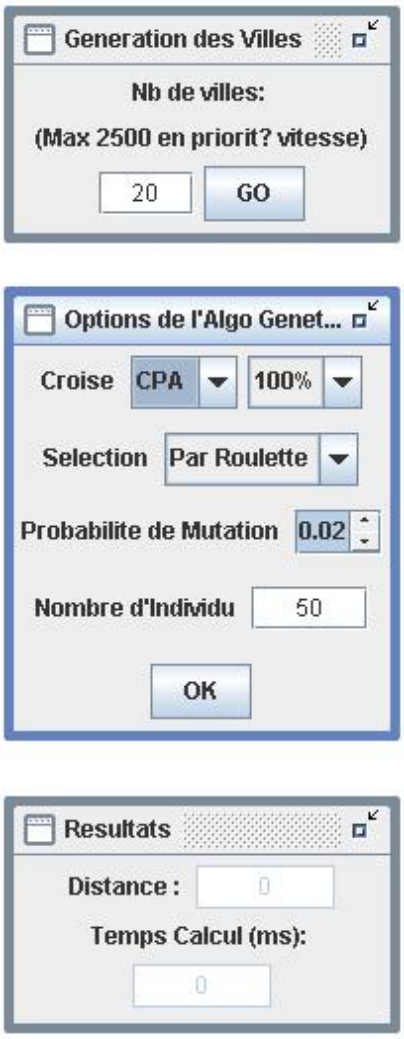


FIG. A.3 – L'interface

Annexe B

Codage des Opérateurs

B.1 Codage de Selection Par Roulette

```
public int[] selectionParRoulette(){

int[] parents = new int[2];

double sommeTotale = moyenneScore()*size();
double bille = genAleatoire.nextDouble()*sommeTotale;
double somme=0.0;

int indice = 0;

for(int i=0; i<2; i++){

somme = 0.0;

while(somme < bille){

indice = genAleatoire.nextInt(size());

somme = somme + individuAt(indice).getScore();
}

parents[i] = indice;
}
```

```
return parents;
}
```

B.2 Codage de Selection Par Rang

```
public int[] selectionParRang(){
int[] parents = new int[2];

int RangTotal = 0;

for(int i=0; i<size(); i++){
RangTotal = RangTotal + i;
}

int bille = genAleatoire.nextInt(RangTotal);
int rang, indice = 0;

for(int i=0; i<2; i++){
rang = 0;

while(rang < bille){

indice = genAleatoire.nextInt(size());

rang = rang + indice;

}
parents[i] = size()-indice-1;
}

return parents;
}
```

B.3 Codage de Selection Par Tournoi

```
public int[] selectionParTournoi(){
```

```
int[] parents = new int[2];
int partenaire;

parents[0] = genAleatoire.nextInt(nbMaxIndividus);
parents[1] = genAleatoire.nextInt(nbMaxIndividus);

partenaire = genAleatoire.nextInt(nbMaxIndividus);
if(individuAt(parents[0]).getScore()<individuAt(partenaire).getScore() )
parents[0] = partenaire;
partenaire = genAleatoire.nextInt(nbMaxIndividus);
if(individuAt(parents[1]).getScore()<individuAt(partenaire).getScore())
parents[1] = partenaire;

return parents;
}
```

B.4 Codage de Croisement CPA

```
public Individu[] croisementCPA (Individu partenaire){

Individu[] fils = new Individu[2];
int[] parcoursFils0 = new int[nbVilles];
int[] parcoursFils1 = new int[nbVilles];
Vector villesAPlacerFils0;
Vector villesAPlacerFils1;
int inf, sup;

villesAPlacerFils0 = new Vector(nbVilles);
villesAPlacerFils1 = new Vector(nbVilles);

for(int i=0; i<nbVilles; i++){
villesAPlacerFils0.addElement(parcours[i]);
villesAPlacerFils1.addElement(partenaire.getPoint(i));

parcoursFils0[i]=-1;
parcoursFils1[i]=-1;
}
```

```
inf = random.nextInt(nbVilles);
sup = inf + random.nextInt(nbVilles - inf);

for (int i=inf ; i < sup ; i++){

parcoursFils0[i] = parcours[i];
villesAPlacerFils0.removeElement((Object)parcours[i]);

parcoursFils1[i] = partenaire.getPoint(i);
villesAPlacerFils1.removeElement((Object)partenaire.getPoint(i));
}

for (int i=0; i < nbVilles ; i++){
if(i == inf)
i = sup;
if(villesAPlacerFils0.contains(partenaire.getPoint(i))){
parcoursFils0[i] = partenaire.getPoint(i);
villesAPlacerFils0.removeElement((Object)partenaire.getPoint(i));
}

if(villesAPlacerFils1.contains(parcours[i])){
parcoursFils1[i] = parcours[i];
villesAPlacerFils1.removeElement(parcours[i]);
}
}

int temp;
for (int i=0; i < nbVilles; i++){
if(i == inf)
i = sup;
if(parcoursFils0[i] == -1){
temp = random.nextInt(villesAPlacerFils0.size());
parcoursFils0[i] = (Integer)villesAPlacerFils0.elementAt(temp);
villesAPlacerFils0.removeElementAt(temp);
}
if(parcoursFils1[i] == -1){
temp = random.nextInt(villesAPlacerFils1.size());
parcoursFils1[i] = (Integer)villesAPlacerFils1.elementAt(temp);
villesAPlacerFils1.removeElementAt(temp);
}
}
```

```
fils[0] = new Individu(parcoursFils0);
fils[1] = new Individu(parcoursFils1);

return fils;

}
```

B.5 Codage de Croisement 1X

```
public Individu[] croisement1X (Individu partenaire){
Individu[] fils = new Individu[2];
int[] enfant1 = new int[nbVilles];
int[] enfant2 = new int[nbVilles];
int posDecoupage = random.nextInt(nbVilles);

for(int i=0; i < posDecoupage; i++){
enfant1[i] = parcours[i];
enfant2[i] = partenaire.getPoint(i);
}

int pos=posDecoupage;

while(pos<nbVilles){
for(int i=0; i<nbVilles; i++){
if(!exist(parcours,posDecoupage,partenaire.getPoint(i))){
enfant1[pos] = partenaire.getPoint(i);
pos++;
}
}
}
pos=posDecoupage;
while(pos<nbVilles){
for(int i=0; i<nbVilles; i++){
if(!exist(partenaire.parcours,posDecoupage,parcours[i])){
enfant2[pos] = parcours[i];
pos++;
}
```

```
}
}

fils[0] = new Individu(enfant1);
fils[1] = new Individu(enfant2);

return fils;
}
```

B.6 Codage de Croisement MPX

```
public Individu[] croisementMPX (Individu partenaire){
Individu[] fils = new Individu[2];
int[] enfant1 = new int[nbVilles];
int[] enfant2 = new int[nbVilles];
int posInf = random.nextInt(nbVilles/2);
int posSup = posInf + random.nextInt(nbVilles - posInf);

for(int i=0; i<nbVilles; i++){
enfant1[i] = -1;
enfant2[i] = -1;
}

for(int i=posInf; i<posSup; i++){
enfant1[i] = parcours[i];
enfant2[i] = partenaire.getPoint(i);
}

// pour creer l'enfant 1
for(int i=0; i<posInf; i++){
if(!exist(enfant1,posSup,partenaire.getPoint(i))){
enfant1[i] = partenaire.getPoint(i);
}
else if(!exist(enfant1,posSup,parcours[i])){
enfant1[i] = parcours[i];
}
else{
for(int j=posInf; j<posSup; j++){
if(!exist(enfant1,posSup,partenaire.getPoint(j))){
```

```
enfant1[i] = partenaire.getPoint(j);
j = posSup;
}
}
}
}

for(int i=posSup; i<nbVilles; i++){
if(!exist(enfant1,i,partenaire.getPoint(i))){
enfant1[i] = partenaire.getPoint(i);
}
else if(!exist(enfant1,i,parcours[i])){
enfant1[i] = parcours[i];
}
else{
for(int j=posInf; j<posSup; j++){
if(!exist(enfant1,i,partenaire.getPoint(j))){
enfant1[i] = partenaire.getPoint(j);
j = posSup;
}
}
}
}


// pour creer l'enfant 2
for(int i=0; i<posInf; i++){
if(!exist(enfant2,posSup,parcours[i])){
enfant2[i] = parcours[i];
}
else if(!exist(enfant2,posSup,partenaire.getPoint(i))){
enfant2[i] = partenaire.getPoint(i);
}
else{
for(int j=posInf; j<posSup; j++){
if(!exist(enfant2,posSup,parcours[j])){
enfant2[i] = parcours[j];
j = posSup;
}
}
}
```

```
}

for(int i=posSup; i<nbVilles; i++){
if(!exist(enfant2,i,parcours[i])){
enfant2[i] = parcours[i];
}
else if(!exist(enfant2,i,partenaire.getPoint(i))){
enfant2[i] = partenaire.getPoint(i);
}
else{
for(int j=posInf; j<posSup; j++){
if(!exist(enfant2,i,parcours[j])){
enfant2[i] = parcours[j];
j = posSup;
}
}
}
}
fils[0] = new Individu(enfant1);
fils[1] = new Individu(enfant2);

return fils;
}
```

B.7 Codage de Croisement OX

```
public Individu[] croisementOX (Individu partenaire){
Individu[] fils = new Individu[2];
int[] enfant1 = new int[nbVilles];
int[] enfant2 = new int[nbVilles];
int posInf = random.nextInt(nbVilles/2);
int posSup = posInf + random.nextInt(nbVilles - posInf);

for(int i=0; i<nbVilles; i++){
enfant1[i] = -1;
enfant2[i] = -1;
}

for(int i=posInf; i<posSup; i++){
```

```
enfant1[i] = parcours[i];
enfant2[i] = partenaire.getPoint(i);
}

for(int i = 0; i < nbVilles-posSup+posInf; i++){
for(int j = 0; j < nbVilles; j++){
if(!exist(enfant1,nbVilles,partenaire.getPoint((posSup+j)%nbVilles))){
enfant1[(posSup+i)%nbVilles] =  partenaire.getPoint((posSup+j)%nbVilles);
j = nbVilles;
}
}
for(int j = 0; j < nbVilles; j++){
if(!exist(enfant2,nbVilles,parcours[(posSup+j)%nbVilles])){
enfant2[(posSup+i)%nbVilles] =  parcours[(posSup+j)%nbVilles];
j = nbVilles;
}
}
}


fils[0] = new Individu(enfant1);
fils[1] = new Individu(enfant2);

return fils;
}
```