



UNIVERSITE HASSAN II DE CASABLANCA
FACULTE DES SCIENCES AIN CHOCK



DEPARTEMENT DE MATHEMATIQUES ET INFORMATIQUE

MÉMOIRE DE PROJET TUTORÉ

Le problème du voyageur de commerce

Réalisé par :

Adib MOHAMED SAFOUNANE
Bellaouali ANAS
Chmarkh HOSSAM

Encadré par :

Prof. LAKHABAB HALIMA

Membres de Jury

Prof. EL MOUMEN Samira
Prof. OUADERHMAN Tayeb
Prof. LAKHBAB Halima



Juin 2021

Table des matières

1 Problème du voyageur de commerce	4
1.1 Histoire	4
1.2 Utilité	5
1.3 Rappel	5
1.4 Modélisation	7
1.5 Variantes	9
2 Méthodes de résolution	10
2.1 Méthodes exactes	10
2.2 Méthodes approximatives	11
2.2.1 Les heuristiques	12
2.2.2 Métaheuristiques :	13
3 Algorithme génétique appliqué au TSP	15
3.1 Principe des algorithmes génétiques	15
3.2 Les avantages algorithmes génétiques	16
3.3 Application sur TSP	17
3.3.1 Représentation (codage) :	17
3.3.2 Génération initiale	18
3.3.3 Fonction d'évaluation	19
3.3.4 Sélection	20
3.3.5 Reproduction	22
3.3.6 Insertion	24
3.3.7 Critère d'arrêt	24
4 Application numérique	25
5 Annexe	36

Remerciement

Nous tenons à remercier tout d'abord, toute l'équipe pédagogique de l'Université Hassan II Faculté des Sciences Aïn Chock, nous pensons avec une gratitude infinie à toute personne ayant contribué de près ou de loin à nous faciliter la tâche d'une façon ou d'une autre par des prières ou autres assistances.

Nos grâces et louanges à Allah le tout puissant, en qui nous mettons toute notre confiance de nous avoir accordé la santé et la volonté d'entamer et d'achever ce mémoire. Sans Sa bonté, ce travail n'aura pas abouti.

Nous tenons à exprimer toute notre reconnaissance à notre encadrant Madame Lakhbab Halima à la fois présente et disponible, qui a su nous guider sur le chemin de notre projet. Nous la remercions de nous avoir encadrés, aidés, aiguillés et conseillés.

Nous adressons nos sincères remerciements aux membres du jury d'avoir accepté de lire et examiner notre travail et d'être venus nous écouter.

Introduction

Faire le tour du monde est une belle expérience, mais le faire d'une façon à visiter un nombre élevé de ville sans passer deux fois par la même ville et revenir chez soi à la fin sera un problème intéressant et difficile à résoudre en même temps.

Cet exemple va nous conduire à un problème que plus tard va être nommé le problème du voyageur de commerce et que nous allons l'abordé par suite. Le concept de ce problème est de visiter plusieurs villes sans passer par la même ville deux fois et avec la moindre distance possible.

L'objectif général de ce document est d'étudier ce problème, son utilité, ses variantes, ainsi que ses différentes méthodes de résolution tout en se concentrant sur l'une des nouvelles méthodes d'optimisation appelé l'algorithme génétique .

Dans le premier chapitre on présente l'histoire du problème, l'utilité de ce problème dans différents domaines, puis les outils mathématiques nécessaires pour comprendre le problème. Dans la suite on s'intéresse aux différentes méthodes de résolution de ce problème d'optimisation combinatoire ainsi que leur avantages et leurs inconvénients pour décider une méthode qu'on va utiliser (l'algorithme génétique). Dans le chapitre d'après on va mettre en valeur cet algorithme tout en expliquant son fonctionnement, citer ses paramètres, ses avantages et l'appliquer sur notre problème.

Chapitre 1

Problème du voyageur de commerce

1.1 Histoire

Le problème du voyageur de commerce (PVC) vient de la traduction anglaise « Traveling Salesman Problem » (TSP), il est l'un des problèmes connus dans le monde de la recherche opérationnelle. Le problème a été formulé mathématiquement au 19ème siècle. Il a été décrit initialement par W.R.Hamilton et Thomas Kirkman. Hamilton l'avait posé sous la forme d'un jeu nommé « Icosian game » en 1857. Le but du jeu est de trouver un cycle hamiltonien le long des bords d'un dodécaèdre tel que chaque sommet est visité une seule fois, et le point départ est le même que celui de l'arrivé. Le TSP a été étudié pour la première fois dans les années 1930 par des mathématiciens un économiste Karl Menger, en suite étudié par des statisticiens : Mohahomlis 1940, Jensen 1942, Cosh 1948 et Julia Robenson 1949. Des recherches approfondies ont été faite à ce propos à l'université de Princeton par Haslen Whitney et Merril Flood. Durant les années 50, un nombre considérable de solutions optimales ont été atteinte grâce à différentes méthodes comme la formulation de George Dantzig Fulkerson et Johnson en 1959, qui a pu atteindre 49 points. En 1956 Merrill Flood a publiée quelques heuristiques générant ainsi de bonnes solutions. En 1957 Barachet a mis en œuvre une méthode de résolution graphique pour le (TSP). Depuis 1954 les chercheurs n'ont pas arrêter d'augmenter le nombre de villes (points).



FIGURE 1.1 – Icosian game

1.2 Utilité

Les applications du problème du voyageur de commerce sont nombreuses pas juste des villes et des distances : le TSP peut se rencontrer dans d'autres contextes : D'une part, certains problèmes d'optimisation de parcours en robotique ou en conception de circuits électroniques (comment percer plusieurs points sur une carte électronique le plus vite possible), l'optimisation de trajectoires de machines outils, ainsi que certains problèmes de séquencement (passage de trains sur une voie, atterrissage d'avions, processus de fabrication en industrie chimique, etc.) s'expriment directement sous forme de TSP.

1.3 Rappel

Généralité sur la Théorie des graphes :

- Un graphe est la donnée d'un ensemble V (vertex) de sommets et un ensemble E (edges) de pairs de sommets appelées arêtes. On écrit $G = (V, E)$.

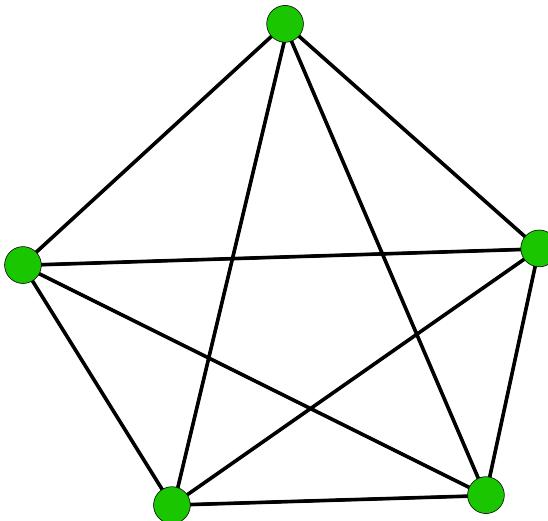


FIGURE 1.2 – graphe complet

- Le degré d'un sommet est le nombre d'arêtes incidentes à ce sommet, on le note $d_g(s)$ ou simplement $d(s)$.

Soit $G = (V, E)$

$$V = \{A, B, C, D, E, F\}$$

$$E = \{(A, B), (B, C), (C, D), (C, E), (C, F)\}$$

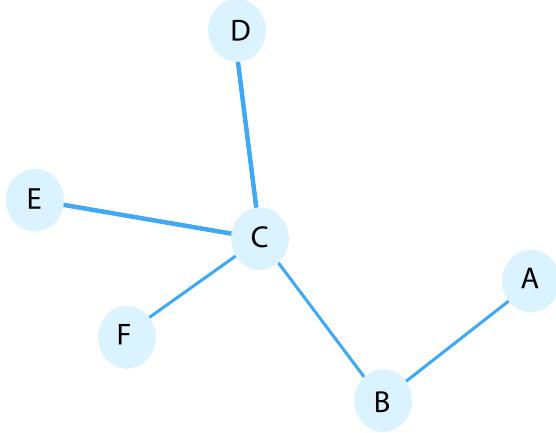


FIGURE 1.3 – Graphe G

$$d(C) = 4$$

$$d(A) = d(D) = d(F) = d(E) = 1$$

$$d(B) = 2$$

- On dit qu'un graphe est complet si tous les arrêtes sont relier deux à deux entre elle.

• Cycle Hamiltoniens

Définition 1.3.1 Soit G un graphe simple. Un cycle hamiltonien est un cycle passant une et une seule fois par tous les sommets de G (le sommet de départ et d'arrivée n'est compté qu'une fois). Le graphe complet à n sommets ($n \geq 3$) a un cycle hamiltonien : les n sommets (dans l'ordre qu'on veut) forment un cycle hamiltonien.

Théorème de Dirac

Soit G un graphe simple à n sommets avec $n \geq 3$. Si pour tout sommet, $d(s) \geq \frac{n}{2}$, alors il existe un cycle hamiltonien dans G .

Notions sur la complexité

Généralement, le temps d'exécution est le facteur majeur qui détermine l'efficacité d'un algorithme, alors la complexité en temps d'un algorithme est le nombre d'instructions nécessaires (affectation, comparaison, opérations algébriques, lecture et écriture, etc.) que comprend cet algorithme pour une résolution d'un problème quelconque.

Définition 1.3.2 Un algorithme en temps polynomial est un algorithme dont le temps de la complexité est en $O(p(n))$, où p est une fonction polynomiale et n est la taille de l'instance (ou sa longueur d'entrée). Si k est le plus grand exposant de ce polynôme en n , le problème correspondant est dit être résoluble en $O(n^k)$ et appartient à la classe P, un exemple de problème polynomial est celui de la connexité dans un graphe.

Définition 1.3.3 La classe NP contient les problèmes de décision qui peuvent être décidés sur une machine non déterministe en temps polynomial. C'est la classe des problèmes qui admettent un algorithme polynomial capable de tester la validité d'une solution du problème. Intuitivement, les problèmes de cette classe sont les problèmes qui peuvent être résolus en énumérant l'ensemble de solutions possibles et en les testant à l'aide d'un algorithme polynomial.

Définition 1.3.4 Un problème est NP-difficile s'il est plus difficile qu'un problème NP-complet, c'est à dire s'il existe un problème NP-complet se réduisant à ce problème par une réduction de Turing.

1.4 Modélisation

Le problème peut être vu mathématiquement comme un graphe complet dont les sommets représentent les villes, et les arêtes représentent les chemins reliant deux villes.

On considère un graphe complet de n sommets $G(V, E)$, $\text{card}(V) = n$.

V : ensemble des sommets

E : ensembles des arêtes

(i, j) est l'arrêt reliant le sommet i et j

d_{ij} représentent la valeur(coût) associé à une arête.

Variables de décisions

Pour deux sommets i et j de V on définit la variable :

$$x_{ij} = \begin{cases} 1 & \text{si on passe de l'arête } (i, j) \\ 0 & \text{sinon} \end{cases}$$

Contraintes

$$1. \quad \begin{cases} \sum_{i=1}^n x_{ij} = 1 & \forall j \in V \\ \sum_{j=1}^n x_{ij} = 1 & \forall i \in V \end{cases}$$

Ces deux contraintes assurent que le voyageur n'entre et ne sort qu'une seule fois par le sommet

$$2. \quad \sum_{ij \in V'} x_{ij} \leq \|V'\| - 1, \quad \forall V' \subset V \text{ et } V' \neq V$$

Cette contrainte élimine les sous-tours au sein d'un tour qui doit commencer et se terminer au même point $\|V'\| = \text{card}(V')$ représente le nombre de sommets faisant partie d'un sous-tour potentiel composé des sommets de l'ensemble V

Exemple

Soit $V = \{4, 5, 6\}$ on a 3 arrêtes liant les trois sommets comme $\|V\| = 3$ alors on se trouve dans le cas d'un sous-tour $4 \rightarrow 5 \rightarrow 6$

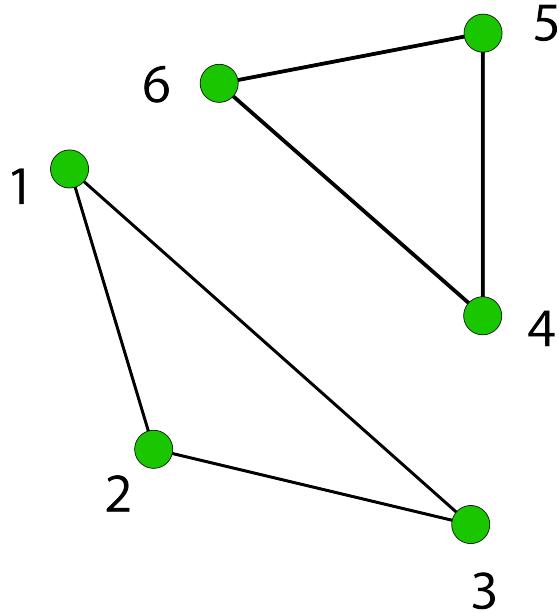


FIGURE 1.4 – exemple d'un sous tour

3. $x_{ij} = \{0, 1\}$ ($ij \in \{1, \dots, n\}$, $i \neq j$)
 Cette contrainte assure que les variables sont binaires.

Fonction objectif

L'objectif est de minimiser la distance totale parcourue pendant le circuit de la tournée

$$\min \sum_{i=1}^n \sum_{j=1}^n d_{ij} x_{ij}$$

Notre problème se modéliser sous la forme :

$$(P) : \begin{cases} \min \sum_{i=1}^n \sum_{j=1}^n d_{ij} x_{ij} \\ \sum_{i=1}^n x_{ij} = 1 & j \in V \\ \sum_{j=1}^n x_{ij} = 1 & i \in V \\ \sum_{ij \in V' - 1} x_{ij} \leq \|V'\|, & \forall V' \subset V \text{ et } V' \neq V \\ x_{ij} = \{0, 1\} & (ij \in \{1, \dots, n\}, i \neq j) \end{cases}$$

1.5 Variantes

Le modèle classique du TSP a été modifié par plusieurs pour l'appliquer à de nombreux problèmes de la vie. Il existe plusieurs problèmes basée sur le TSP a savoir ; TSP symétrique (STPS), TSP asymétrique (ATSP) et TSP multiple (MTPS). Ces variantes peuvent être décrites comme suit :

- STSP : soit d_{ij} la distance entre les villes i et j . Si $d_{ij} = d_{ji}$ alors le trajet peut être évalué dans les deux directions pour le même coût . le STSP est le même que le TSP et consiste à trouver un chemin de longueur minimal qui visite chaque ville une fois.
- ATSP : si d_{ij} est différent de d_{ji} par au moins un sommet, alors le TSP devient un ATSP.
- MTSP : est un TSP avec plusieurs voyageurs de commerce .

Chapitre 2

Méthodes de résolution

Pour résoudre le TSP, on trouve des algorithmes déterministes (exactes) et les algorithmes d'approximation (heuristiques et métaheuristiques).

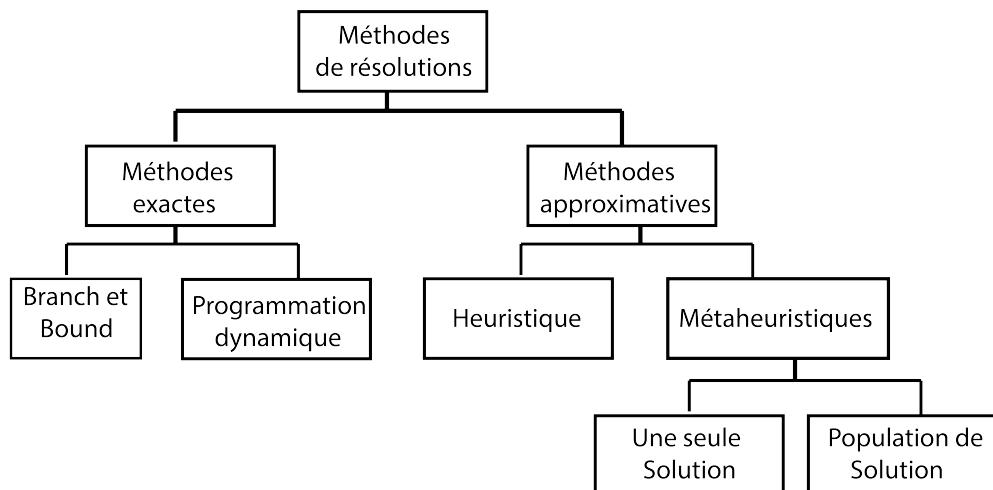


FIGURE 2.1 – les Méthodes de résolution

2.1 Méthodes exactes

Les algorithmes exactes permettent de trouver la solution optimale mais leur complexité et d'ordre exponentiel : Les méthodes exactes explorent de façon systématique l'espace de recherche. Le TSP est plus compliqué qu'il n'y paraît ; on ne connaît pas de méthodes de résolution exactes permettant d'obtenir des solutions en un temps raisonnable pour de grandes instances (grand nombre de villes) du problème. Pour ces grandes instances on se retrouve face à une explosion combinatoire.

Pour n sommets il existe au total $n!$ solutions à explorer. On a donc $\frac{1}{2} (n-1)!$ chemins candidats à considérer.

Parmi les algorithmes les plus utilisés on cite la méthode de séparation et évaluation et la méthode des plans sécants.

Branch and Bound (Séparation et évaluation)

L'algorithme de séparation et évaluation, plus connu sous son appellation anglaise Branch and Bound (Land et Doig 1960), repose sur une méthode arborescente de recherche d'une solution optimale par séparations et évaluations, en représentant les états solutions par un arbre d'états, avec des noeuds, et des feuilles. Le branch-and-bound est basé sur trois axes principaux :

- L'évaluation,
- La séparation,
- La stratégie de parcours.

1. L'évaluation : L'évaluation permet de réduire l'espace de recherche en éliminant quelques sous ensembles qui ne contiennent pas la solution optimale. L'objectif est d'essayer d'évaluer l'intérêt de l'exploration d'un sous-ensemble de l'arborescence. Le branch-and-bound utilise une élimination de branches dans l'arborescence de recherche de la manière suivante : la recherche d'une solution de coût minimal, consiste à mémoriser la solution de plus bas coût rencontré pendant l'exploration, et à comparer le coût de chaque noeud parcouru à celui de la meilleure solution. Si le coût du noeud considéré est supérieur au meilleur coût, on arrête l'exploration de la branche et toutes les solutions de cette branche seront nécessairement de coût plus élevé que la meilleure solution déjà trouvée.
2. La séparation : La séparation consiste à diviser le problème en sous-problèmes. Ainsi, en résolvant tous les sous-problèmes et en gardant la meilleure solution trouvée, on est assuré d'avoir résolu le problème initial. Cela revient à construire un arbre permettant d'énumérer toutes les solutions. L'ensemble de noeuds de l'arbre qu'il reste encore à parcourir comme étant susceptibles de contenir une solution optimale, c'est-à-dire encore à diviser, est appelé ensemble des noeuds actifs.
3. La stratégie de parcours
 - La largeur d'abord : Cette stratégie favorise les sommets les plus proches de la racine en faisant moins de séparations du problème initial. Elle est moins efficace que les deux autres stratégies présentées.
 - La profondeur d'abord : Cette stratégie avantage les sommets les plus éloignés de la racine (de profondeur la plus élevée) en appliquant plus de séparations au problème initial. Cette voie mène rapidement à une solution optimale en économisant la mémoire
 - Le meilleur d'abord : Cette stratégie consiste à explorer des sous-problèmes possédant la meilleure borne. Elle permet aussi d'éviter l'exploration de tous les sous-problèmes qui possèdent une mauvaise évaluation par rapport à la valeur optimale.

2.2 Méthodes approximatives

Les algorithmes d'approximation permettent de trouver une solution approchée de la solution optimale en un temps raisonnable. Ils sont généralement utilisés pour des

problèmes de grande taille.

Pour faire face aux problèmes de l'explosion combinatoire les algorithmes approximatifs utilisent des processus aléatoires.

2.2.1 Les heuristiques

Une heuristique est une méthode d'optimisation simple et rapide ayant pour but de trouver une valeur approximative à un problème donné. Les heuristiques ont la particularité de traiter un problème avec le minimum d'information donnée mais par contre elle n'offre aucune garantie que le minimum global est atteint.

Parmi les algorithmes approximatifs on trouve l'algorithme du plus proche voisin (Nearest Neighbour) :

- L'algorithme du plus proche voisin :

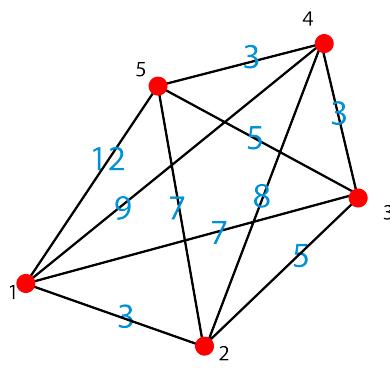
C'est l'un des plus simples on peut le diviser en deux grandes étapes. La première étape repose sur le choix aléatoire d'une première ville, et l'étape suivante consiste à se déplacer de ville en ville en choisissant la ville la plus proche , c'est-à-dire en sélectionnant la prochaine ville telle que la distance entre la ville courante et la prochaine ville soit le plus minimal, et ça jusqu'à avoir visité toutes les villes. Il faut enfin revenir à la ville choisie au départ, pour obtenir un cycle.

Nearest Neighbour (NN)

[1] Choisir un sommet $v_1 \in V$ Poser
 $k \leftarrow 1$
[2] Tant que $k < n$
 $k \leftarrow k + 1$
choisir v_k dans $V \setminus \{v_1, v_2, \dots, v_{k-1}\}$
qui minimise c_{v_{k-1}, v_k}
Retourner le tour (v_1, v_2, \dots, v_n) .

Exemple :

Soit le graphe suivant :



2

On prend 3 comme point de départ de l'algorithme ce qui nous donne le résultat

FIGURE 2.2 – graphe

suivant.

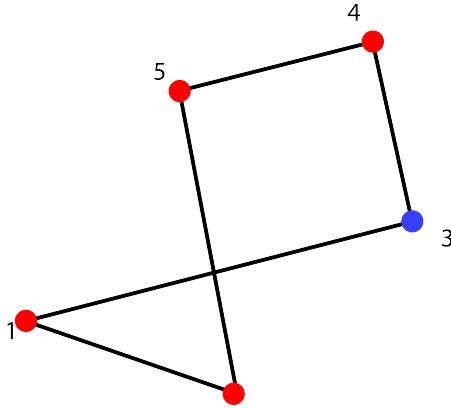


FIGURE 2.3 – résultat

2.2.2 Métaheuristiques :

Les métaheuristiques ne sont que des heuristiques à la base mais elles ont la particularité d'être adaptable à plusieurs problèmes sans effectuer un changement majeur sur l'algorithme.

Parmi les algorithmes métaheuristiques on trouve l'algorithme génétique, et les algorithmes par essaims particulaires.

Optimisation par essaims particulaires

Cet algorithme s'inspire à l'origine du monde du vivant. Il s'appuie notamment sur un modèle développé par Craig Reynolds à la fin des années 1980, permettant de simuler le déplacement d'un groupe d'oiseaux. Une autre source d'inspiration, revendiquée par les auteurs, James Kennedy et Russel Eberhart, est la socio-psychologie 1. Cette méthode d'optimisation se base sur la collaboration des individus entre eux. Elle a d'ailleurs des similarités avec les algorithmes de colonies de fourmis, qui s'appuient eux aussi sur le concept d'auto-organisation. Cette idée veut qu'un groupe d'individus peu intelligents peut posséder une organisation globale complexe. Ainsi, grâce à des règles de déplacement très simples (dans l'espace des solutions), les particules peuvent converger progressivement vers un minimum global. Cette métaheuristique semble cependant mieux fonctionner pour des espaces en variables continues. Au départ de l'algorithme chaque particule est donc positionnée (aléatoirement ou non) dans l'espace de recherche du problème. Chaque itération fait bouger les particules en fonction de 3 composantes :

-
1. Sa vitesse actuelle V_k
 2. Sa meilleure solution P_i
 3. La meilleure solution obtenue dans son voisinage P_g

Cela donne l'équation de mouvement suivante :

- $V_{k+1} = \omega V_k + b_1(P_i - X_k) + b_2(P_g - X_k)$
- $X_{k+1} = X_k + V_{k+1}$

Avec :

- X_k sa position actuelle
- ω inertie
- b_1 tiré aléatoirement dans $[0, \phi_1]$
- b_2 tiré aléatoirement dans $[0, \phi_2]$

Chapitre 3

Algorithme génétique appliqué au TSP

Introduction

Les algorithmes génétiques (AG) font partie de la famille des algorithmes évolutifs. Ils s'inspirent du credo de la nature (survie pour l'individu le plus adapté à l'environnement). Ces algorithmes s'inspirent de la théorie de l'évolution naturelle des espèces. Cité d'abord par HOLLAND qu'a développer les principe fondamentale puis GOLDBERG qui les a utiliser pour résoudre des problème d'optimisation.

3.1 Principe des algorithmes génétiques

Généralement les algorithmes génétiques sont basé sur les étapes suivantes :

- Choisir le type de codage des solutions.
- Générer une population initiale de solution de taille fini N.
- Définir une fonction d'évaluation (fitness) pour évaluer les solutions.
- Sélectionner aléatoirement des solutions (parents).
- Générer une nouvelle génération des solutions à l'aide des opérateurs suivants :
 1. Opérateur de Croisement : Générer de nouvelles solutions(enfants) par accouplement des parents

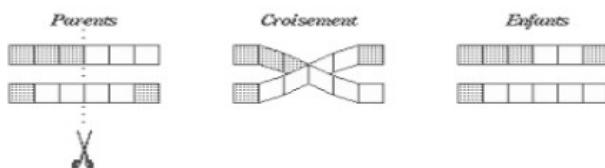


FIGURE 3.1 – croisement

2. Opérateur de Mutation : changer aléatoirement la structures d'une solution afin d'évité les optimums locaux.

- Regrouper les solutions parents et enfants.

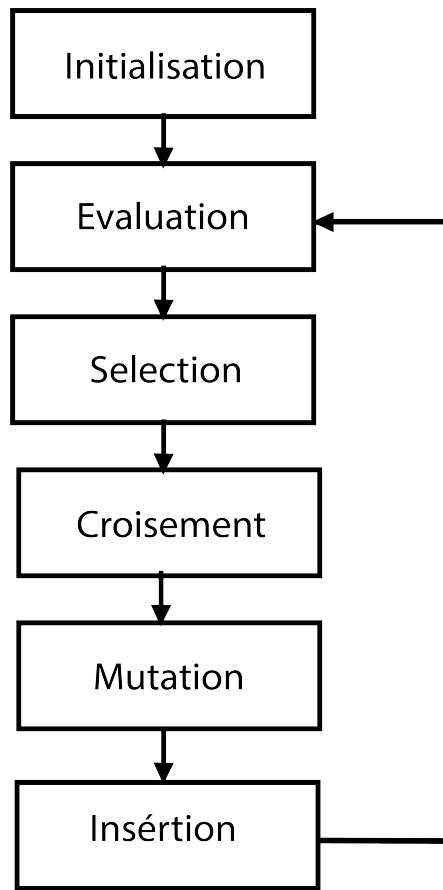


FIGURE 3.2 – Organigramme AG

3.2 Les avantages algorithmes génétiques

- L'utilisation de l'algorithme génétique ne nécessite que l'évaluation de la fonction fitness, on n'a aucun intérêt à utiliser les propriétés de la fonction objectif elle-même (continuité ,dérivabilité etc ...). Ce qui donne plus de souplesse en large domaine d'application.
- L'AG traite plusieurs solutions à la fois (Population) contre les algorithmes classiques qui traitent une seul solution à chaque itération.
- Les règles de transition probabilistes permettent dans certaines situations d'éviter les optimums locaux et se diriger vers un optimum global.

3.3 Application sur TSP

3.3.1 Représentation (codage) :

La simulation de la théorie de l'évolution exige une représentation des solutions (individus) sous forme d'un chromosome, que l'on peut décrire mathématiquement sous forme d'un vecteur du chiffre ou alphabet etc ... (ça dépend du probablement donnée).

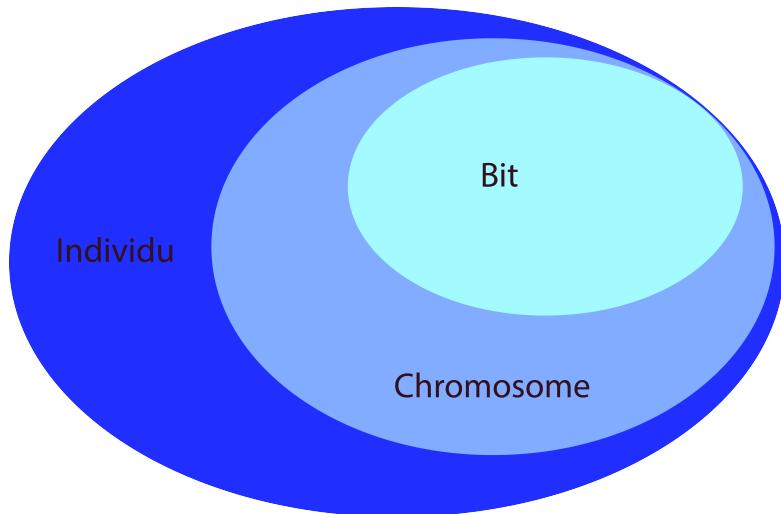


FIGURE 3.3 – Niveaux d'organisation de notre algorithme génétique.

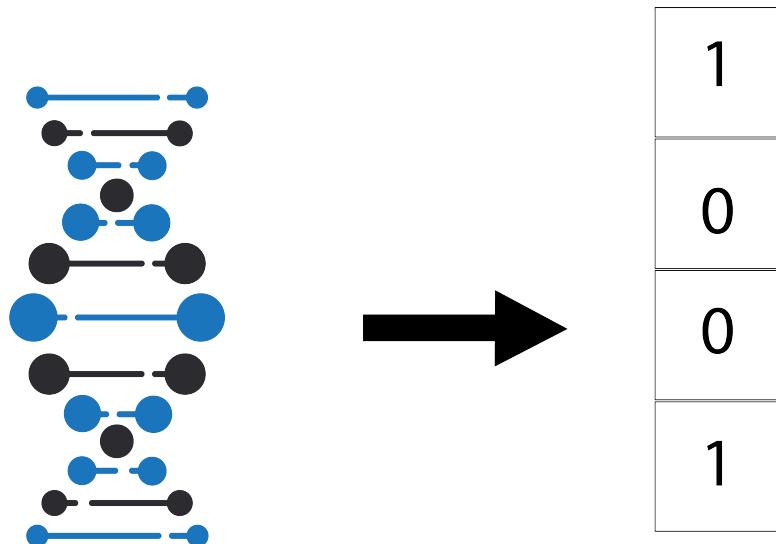


FIGURE 3.4 – Codage d'un chromosome

On vas se limiter aux types des représentations convenables à notre problème (TSP).

• La représentation adjacente :

La représentation adjacente représente la tournée comme une liste de N villes commençant par la ville 1. La ville j est listée à la position i si est seulement si le voyageur de commerce se dirige de la ville i à la ville j.

Exemple :

2	4	8	3	9	7	1	5	6
---	---	---	---	---	---	---	---	---

Représentation adjacente de la tournée

$$1 \rightarrow 2 \rightarrow 4 \rightarrow 3 \rightarrow 8 \rightarrow 5 \rightarrow 9 \rightarrow 6 \rightarrow 7 \rightarrow 1.$$

Chaque solution a une représentation unique mais parfois on trouve quelques listes adjacentes représentent un parcourt non permis par exemple :

2	4	8	1	9	3	5	7	6
---	---	---	---	---	---	---	---	---

Ce chromosome n'est pas une solution de TSP car il contient un sous tour
 $1 \rightarrow 2 \rightarrow 4 \rightarrow 1$.

• La représentation chemin :

Un circuit est codé par un tableau des entiers qui représente le successeur et le prédécesseur de chaque ville.

Exemple :

6	2	4	1	5	3	7
---	---	---	---	---	---	---

Ainsi, le tableau représente le circuit suivant : le point de départ est la ville 6 qui est aussi la ville d'arrivée , en passant par les villes selon leur ordre d'apparition dans le tableau :

$$6 \rightarrow 2 \rightarrow 4 \rightarrow 1 \rightarrow 5 \rightarrow 3 \rightarrow 7 \rightarrow 6.$$

La représentation la mieux adaptée à notre problème est la représentation chemin.

3.3.2 Génération initiale

Dans les procédures de GA, une connaissance primordiale des individus de bonne qualité comme points d'initialisation conditionne la rapidité de la convergence vers l'optimum. Si on ne dispose pas de ces connaissances on génère aléatoirement des individus. Les tirages sont réalisés en respectant les contraintes. Par exemple la matrice suivante décrit une génération initiale de 4 individus qui représentent quatre chemins passant par sept villes.

6	2	4	1	5	3	7
3	2	1	7	6	5	4
6	4	5	1	2	7	3
5	1	3	6	4	7	2

3.3.3 Fonction d'évaluation

La fonction de performance -qu'on appelle aussi fonction d'adaptation, ou fonction fitness. La construction d'une fonction d'adaptation appropriée à partir de la fonction objectif est très importante pour obtenir un bon fonctionnement des AG.

- Principe :

L'évaluation est l'analyse des individus pour analyser si une solution est valide. Pour ceci, nous utilisons une fonction de coût, ou d'erreur, provenant de la fonction objectif afin de définir le score d'adaptation des individus lors du processus de sélection.

- Application :

On associe une valeur de performance à chaque individu ce qui offre la possibilité de le comparer à d'autres individus et permet à l'algorithme génétique de déterminer qu'un individu sera sélectionné pour être reproduit ou pour déterminer s'il sera remplacé.

- Évaluation pour le TSP :

Les algorithmes génétiques travaillent sur la maximisation d'une fonction positive. Or dans beaucoup de problèmes, l'objectif est de minimiser une fonction coût. Et même, dans un problème de maximisation rien ne garantit que la fonction objectif reste positive pour tout individu X. Ainsi, il est souvent nécessaire de transformer la fonction objectif en une fonction appelée fonction d'adaptation. La fonction d'adaptation pour le TSP est :

$$f_a(X) = f_{max} - f(X)$$

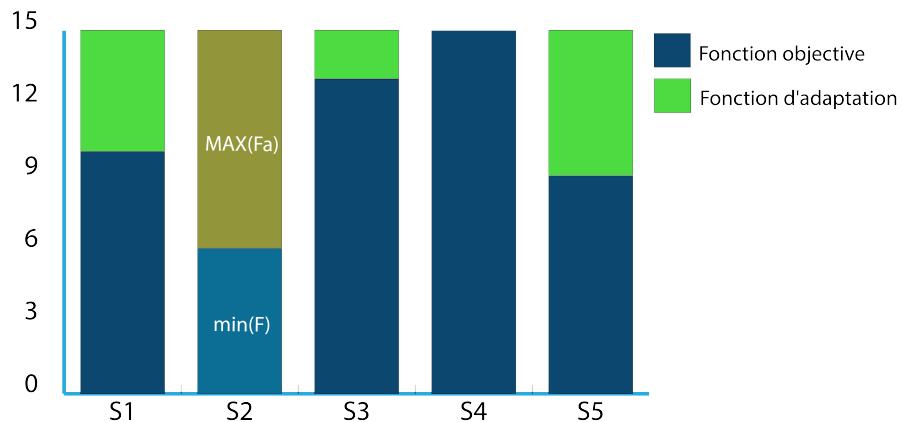


FIGURE 3.5 – L'évaluation par la fonction objectif et la fonction fitness

3.3.4 Sélection

Sélection naturelle (définition de la biologie) : un processus dans la nature dans lequel les organismes possédant certaines caractéristiques génotypiques qui les rendent mieux adaptés à un environnement ont tendance à survivre. La sélection : consiste à choisir aléatoirement les individus les mieux adaptés à survivre en utilisant la fonction d'adaptation. On trouve deux types de sélection :

- 1. La sélection pour la reproduction :

On l'appelle tout simplement l'opération de sélection, et elle permet de choisir les individus qui participent à une reproduction . Cette opération choisit, généralement, les individus ayant le meilleurs scores d'adaptation pour produire les enfants les plus performants.

- 2. La sélection pour le remplacement :

On l'appelle tout simplement l'opération de remplacement, et elle choisit les individus les plus faibles pour être remplacés par les nouveaux.

La sélection par tournoi :

Un nombre p d'individus est sélectionné parmi les N individus de la population. Le meilleur, c'est à dire celui dont la fonction d'adaptation est la plus élevée, est sélectionné. Ce procédé est répété jusqu'à obtenir N individus. Il est tout à fait possible que certains individus participent à plusieurs tournois. Ils ont donc droit d'être copiés plusieurs fois, ce qui favorise la pérennité de leurs gènes. En général quatre individus sont sélectionnés pour chaque tournoi, $p=4$.

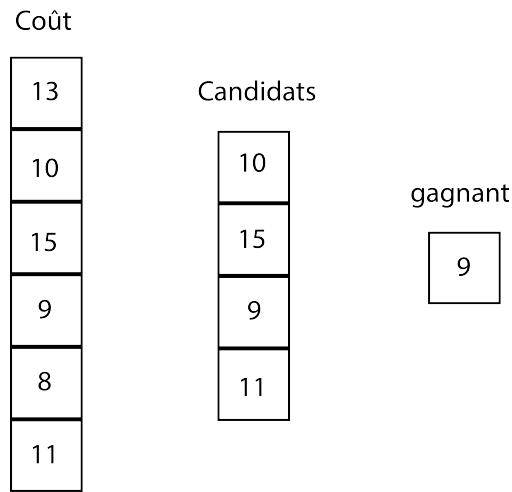


FIGURE 3.6 – Sélection par tournoi

La sélection roulette de loterie :

La sélection proportionnelle ou sélection par roue de loterie consiste à dupliquer chaque individu proportionnellement à la valeur de la fonction d'adaptation. Ceci peut être effectué aisément en procédant à des tirages aléatoires consécutifs où chaque individu a une probabilité d'être sélectionnée égale à

$$p_s(i) = \frac{f_a(i)}{\sum_{j=1}^N f_a(j)}$$

où i désigne l'individu.

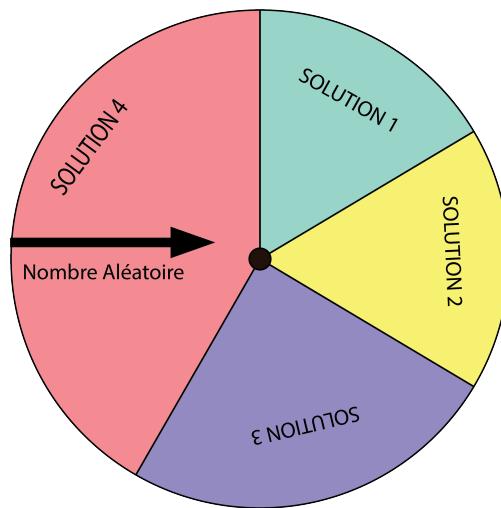


FIGURE 3.7 – Sélection par roulette

3.3.5 Reproduction

La reproduction est un processus biologique qui permet la production de nouveaux organismes d'une espèce à partir d'individus préexistants de cette espèce. La reproduction augmente le nombre des espèces et permet de transmettre et de perpétuer leurs qualités génotypiques essentielles aux générations suivantes. La reproduction est définie dans deux grandes étapes croisement et mutation.

Le croisement

Est une opération qui permet l'échange d'information entre les individus (solutions). Il y'a différents types de croisement applicable à notre problème : PMX (partially mapped crossover) OX (order crossover).

- PMX : Le croisement partiellement cartographié (PMX) a été proposé par Goldberg et Lingle. Après avoir choisi deux points de coupure aléatoires sur les parents pour construire une progéniture, la partie entre les points de coupure, la chaîne d'un parent est mappée sur la chaîne de l'autre parent et les informations restantes sont échangées.

Exemple : Soit deux parents de taille 8 (2 chemin entre 8 villes)

$$P_1 = (3 \ 4 \ 8 \ 2 \ 7 \ 1 \ 6 \ 5).$$

$$P_2 = (4 \ 2 \ 5 \ 1 \ 6 \ 8 \ 3 \ 7).$$

On prend deux nombres aléatoires entre 1 et la longueur du chromosome par exemple 3 et 6 alors la première coupure sera entre la troisième et la quatrième ville , et la deuxième sera entre la sixième et la septième ville .

Après on crée les deux enfants par les sections entre les coupures des parents

$$E_1 = (XXX | 168 | XX).$$

$$E_2 = (XXX | 271 | XX).$$

Puis on continue le remplissage de chaque enfant par les bits de l'autre parent qui n'appartient pas a cet enfant , gardant la même position des bits dans le chromosome parent .

$$E_1 = (34X | 168 | X5).$$

$$E_2 = (4X5 | 271 | 3X).$$

Dans le premier parent la ville numéro 8 est placée dans la troisième position , et il existe déjà dans le premier enfant à la 6ème position, donc on regarde la valeur dans la 6ème position au 2ème enfant $8 \rightarrow 1$ comme 1 existe déjà dans le premier enfant on prend du 2ème enfant la ville de même position que 1 dans l'enfant 1 (voir la figure).

$$E_1 = (3\ 4\ 2 | 1\ 6\ 8 | X\ 5) .$$

$\uparrow \downarrow \uparrow \downarrow$

$$E_2 = (4\ X\ 5 | 2\ 7\ 1 | 3\ X) .$$

On répète ce processus pour tout les points restants.

$$E_1 = (3\ 4\ 2 | 1\ 6\ 8 | 7\ 5) .$$

$$E_2 = (4\ 8\ 5 | 2\ 7\ 1 | 3\ 6) .$$

- OX : L'ordre croisé (OX) a été proposé par Davis . Il crée une descendance en choisissant un sous-tour d'un parent et en préservant l'ordre relatif des bits de l'autre parent.

Exemple :

$$\left\{ \begin{array}{l} P = (1\ 2\ 3 | \color{red}{4\ 5\ 6\ 7} | 8\ 9) \\ P = (\color{blue}{2\ 4\ 6\ 7\ 9\ 1\ 3} | \color{blue}{8\ 5}) \end{array} \right. \xrightarrow{\text{OX}} E = (\color{blue}{2\ 9\ 1} | \color{red}{4\ 5\ 6\ 7} | \color{blue}{3\ 8})$$

La mutation

La mutation est un phénomène rare mais permet d'explorer de nouvelles zones dans l'espace de recherche et aide l'algorithme génétique à possiblement aller vers une solution optimale globale, sans rester dans une solution optimale locale.

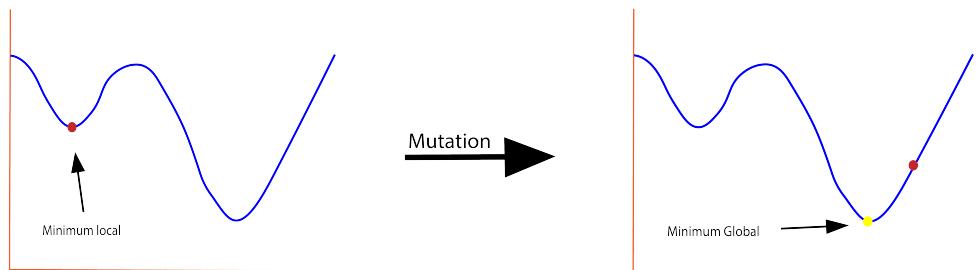
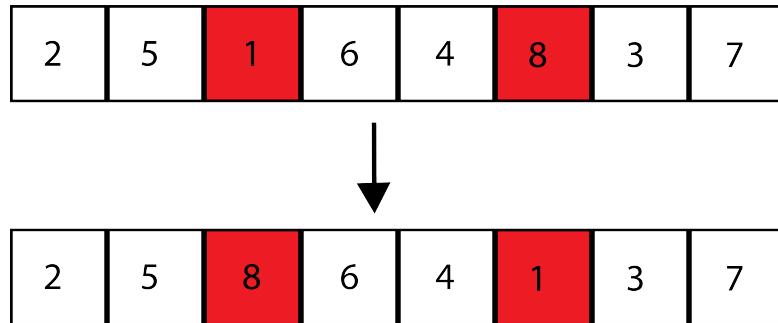


FIGURE 3.8 – Effet de la mutation

Le rôle de cet opérateur est de modifier aléatoirement, avec une certaine probabilité, la valeur d'un composant de l'individu.

Exemple :Mutation d'un chromosome.



3.3.6 Insertion

Finalement on est dans la phase d'insertion, où on doit créer la nouvelle génération des individus qui sera la population initiale de l'itération suivante, généralement on doit mélanger les enfants et les parents, mais les détails varient selon les résultats désirés. Pour notre algorithme on a choisi de prendre la totalité des enfants et complété ce qui nous reste aléatoirement par les parents, et on a ajouté l'approche d'élitisme qui permet de garder à la fin l'individu avec la plus grande fitness dans la nouvelle génération pour assurer la convergence vers le minimum global.



FIGURE 3.9 – insertion

3.3.7 Critère d'arrêt

Nous proposons qu'un nombre de générations soit à préciser comme critère d'arrêt. À défaut de cela, c'est la convergence de la population qui détermine l'arrêt de l'algorithme.

Chapitre 4

Application numérique

Dans ce chapitre nous allons faire des tests pour vérifier numériquement l'efficacité de l'algorithme génétique. Pour déterminer une bonne solution à l'aide de l'algorithme génétique, il faut bien choisir les paramètres. Probabilité d'utiliser la roulette comme méthode de sélection.

Probabilité d'utiliser le méthode de tournoi pour la sélection.

Probabilité d'utiliser le croisement par la méthode OX.

Probabilité d'utiliser le croisement par la méthode PMX.

Probabilité d'utiliser le facteur de mutation sur un individu.

Les testes sont effectués sur 13 villes .



La matrice du coût de l'exemple est la suivante :

0	2451	713	1018	1631	1374	2408	213	2571	875	1420	2145	1972
2451	0	1745	1524	831	1240	959	2596	403	1589	1374	357	579
713	1745	0	355	920	803	1737	851	1858	262	940	1453	1260
1018	1524	355	0	700	862	1395	1123	1584	466	1056	1280	987
1631	831	920	700	0	663	1021	1769	949	796	879	586	371
1374	1240	803	862	663	0	1681	1551	1765	547	225	887	999
2408	959	1737	1395	1021	1681	0	2493	678	1724	1891	1114	701
213	2596	851	1123	1769	1551	2493	0	2699	1038	1605	2300	2099
2571	403	1858	1584	949	1765	678	2699	0	1744	1645	653	600
875	1589	262	466	796	547	1724	1038	1744	0	679	1272	1162
1420	1374	940	1056	879	225	1891	1605	1645	679	0	1017	1200
2145	357	1453	1280	586	887	1114	2300	653	1272	1017	0	504
1972	579	1260	987	371	999	701	2099	600	1162	1200	504	0

(4.1)

La distance optimale relativ à cet exemple est 7293 miles



Notation

Dans ce qui suit les Box-plot et les tableaux sont fait à l'aide du logiciel R.
Les graphes sont fait sur Matlab.

- Le 0e quartile est la donnée de la série qui sépare les 0% inférieurs des données, noté Min.
- Le premier quartile, noté Q1, sépare le premier quart des données du reste des données.
- Le deuxième quartile, noté Median, sépare la distribution en deux parties égales.

-
- Le troisième quartile, noté Q3, sépare les trois premiers quartils des données du reste des données.
 - le 4e quartile est la donnée de la série qui sépare les 0% supérieurs des inférieurs des données, noté Max

Sélection

Dans cette partie on va comparer les moyennes des 40 testes faites sur les méthodes de sélection que nous avons conçus (roulette , tournoi).

Le nombre d'itération sera 50, 100, 150 pour les deux méthodes

Roulette

Nombre d'itérations	50	100	150
Min	7295	7293	7590
Q1	8086	8071	7946
Median	8328	8248	8206
Moyenne	8279	8246	8227
Q3	8547	8426	8445
Max	8890	9121	9196

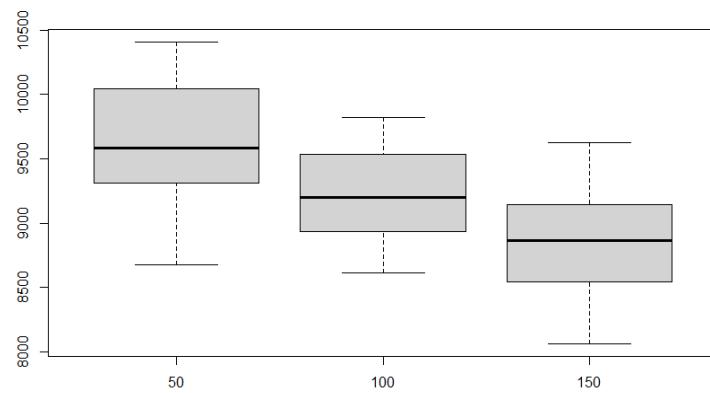


FIGURE 4.1 – Box-plot roulette

On remarque que plus les itérations augmentent, la moyenne des résultats diminue. De plus grâce à la nature aléatoire de l'algorithme , l'augmentation des itérations n'implique pas forcément que le résultat soit meilleur . Dans notre exemple on a trouvé le minimum global avec 100 itérations non pas avec 150 itérations

Tournoi

Nombre d'itérations	50	100	150
Min	7534	7293	7295
Q1	8665	8066	7919
Median	8902	8438	8206
Moyenne	8954	8381	8150
Q3	9373	8726	8330
Max	10244	9193	9182

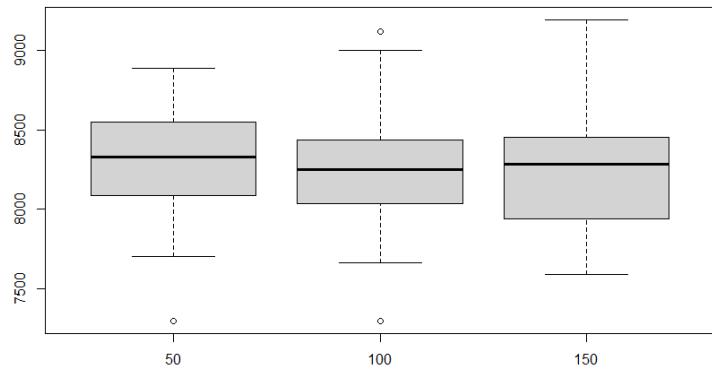
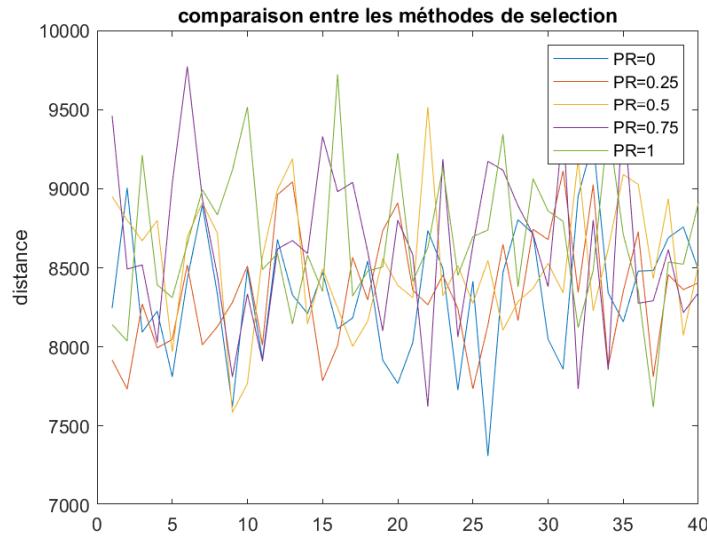


FIGURE 4.2 – Box-plot tournoi

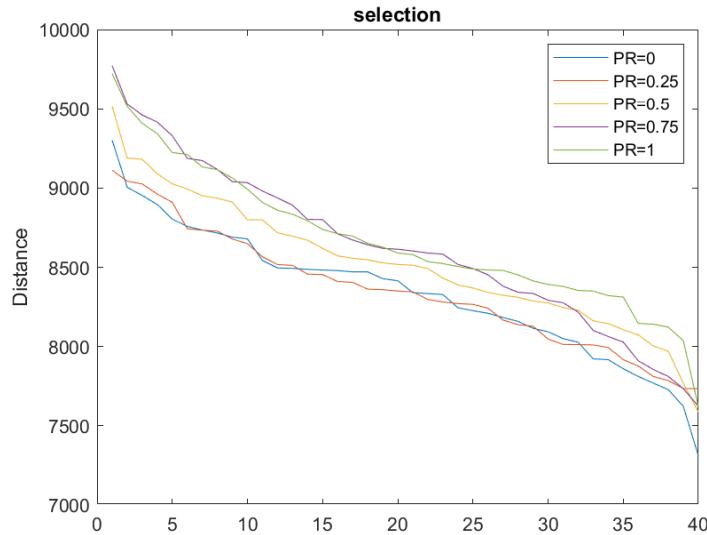
On remarque que généralement le nombre d'itération n'as pas une grande influence sur les résultats.

Les deux méthodes de sélection en même temps

Notre algorithme donne aussi la possibilité de melanger les deux types de sélection. Si on pose PR=0,2 c'est à dire que la roulette sera utilisé deux fois chaque 10 itérations. Dans ce qui suit on va fixer le nombre d'itération à 150 et varier PR entre {0, 0.25, 0.5, 0.75, 1}



Les résultats sont aléatoires donc le graphe n'est pas lisible. Le graphe suivant comporte les mêmes résultats mais ils sont ordonnés .



On remarque que l'utilisation d'une seule méthode pendant tout l'algorithme donne de meilleurs résultats. par suite

$$PR = \begin{cases} 1 & \text{si le nombre d'itération } \geq 150 \\ 0 & \text{sinon} \end{cases}$$

Croisement

Dans cette partie on vas comparer entre les deux méthodes de croisement PMX et OX.

Le nombre d'itération sera 50, 100, 150 pour les deux méthodes.

Croisement PMX

Nombre d'itérations	50	100	150
Min	7310	7293	7312
Q1	8011	7665	8120
Median	8364	7939	8366
Moyenne	8335	7907	8394
Q3	8662	8140	8736
Max	9161	8699	9425

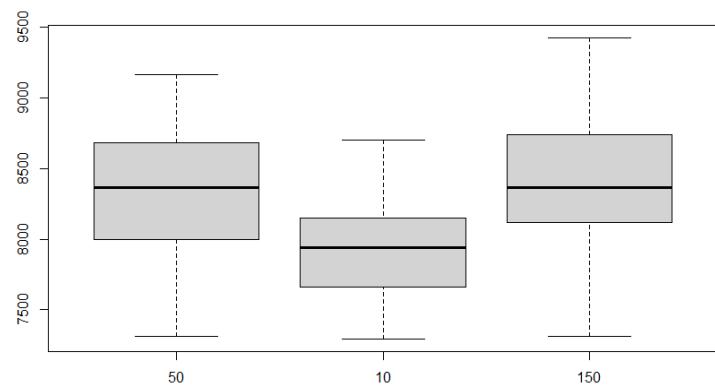


FIGURE 4.3 – Box-plot PMX

Croisement OX

Nombre d'itérations	50	100	150
Min	7293	7295	7598
Q1	8138	7612	8010
Median	8438	7690	8302
Moyenne	8367	7770	8339
Q3	8608	8023	8554
Max	9197	8312	9569

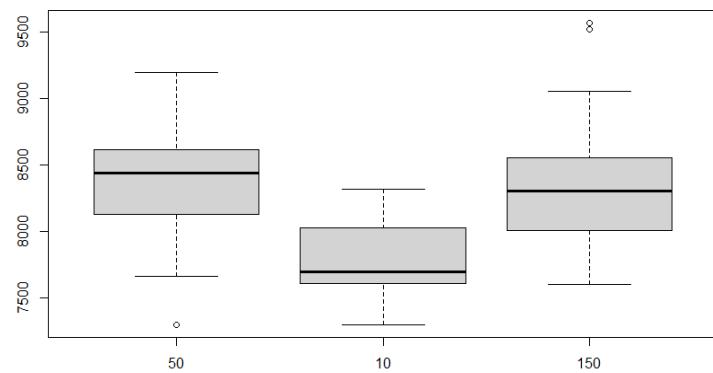
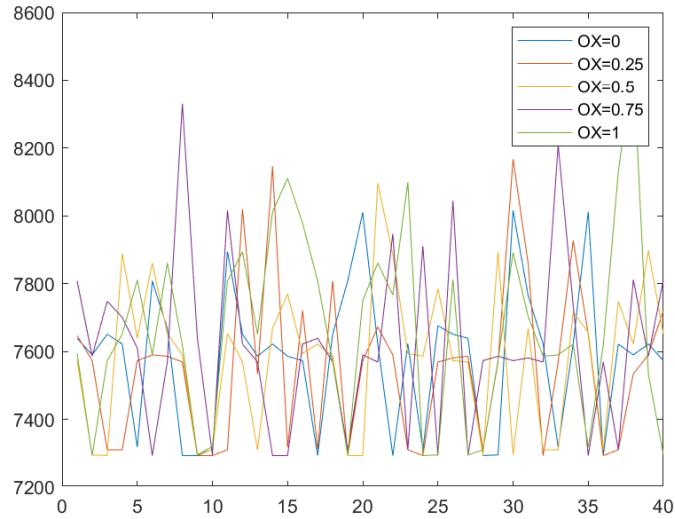


FIGURE 4.4 – Box-plot OX

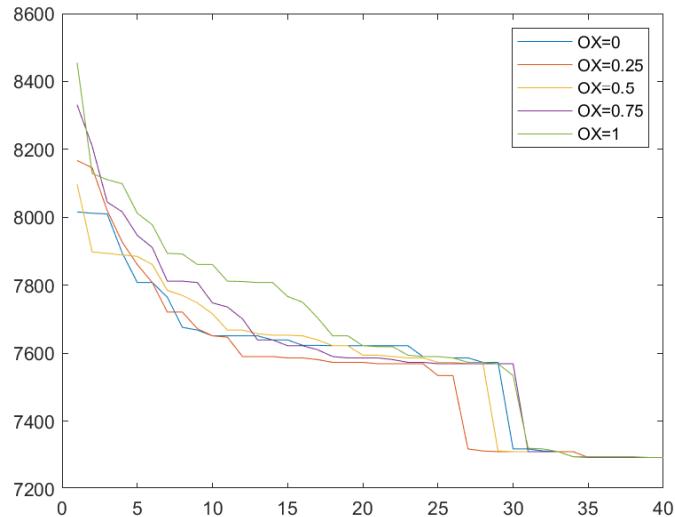
Remarque

Le nombre d'itération n'a pas une grande influence sur les résultats.

Les deux méthodes de croisement en même temps



les résultats sont aléatoires donc le graphe n'est pas lisible le graphe suivant comporte les mêmes résultats mais ils sont ordonné .



Les résultats obtenus sont proches, donc on peut choisir entre l'une des deux ou bien le mélange donc dans la suite OX =0.5

Mutation

Dans cette partie on vas tester le paramètre de mutation . la probabilité du mutation varie entre 0 ,0.01 ,0.05 ,0.1 ,0.15 ,0.2.

Probabilité du Mutation	0	0.01	0.05	0.1	0.15	0.2
Min	7293	7293	7293	7293	7293	7293
Q1	7900	7293	7293	7312	7534	7725
Median	8339	7569	7423	7586	7651	8122
Moyenne	8339	7521	7458	7577	7729	8086
Q3	8656	7622	7586	7651	7988	8394
Max	10265	8045	7890	8528	8343	9189

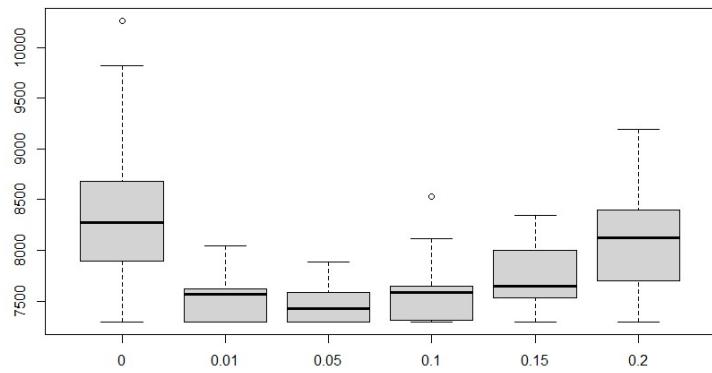
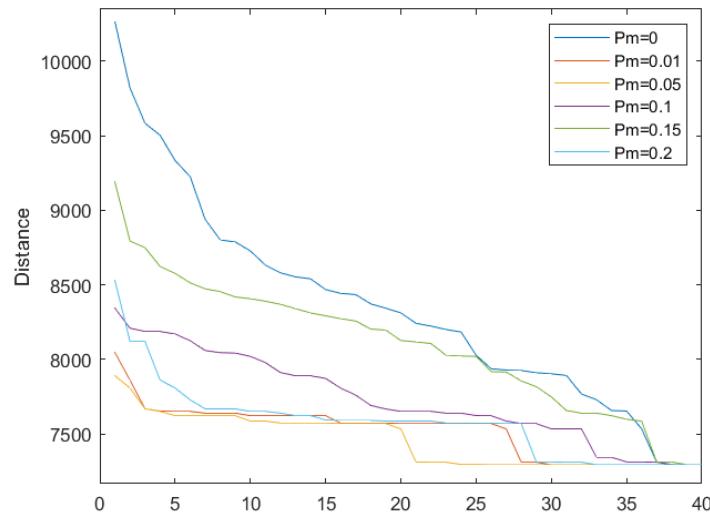


FIGURE 4.5 – Box-plot mutation



On peut conclure que la probabilité de permutation a un grand effet sur la convergence de l'AG comme on a déjà vu, les résultats obtenu par l'algorithme où $Pm = 0$ sont plus grand que les autres. D'après les résultats obtenus on remarque que

si P_m est entre 0.01 et 0.1 ça sera mieux pour notre cas.

Exemple

Dans cet exemple en se référant sur 9 villes du Maroc. La matrice des distances est :

```
Cout_mat = 9x9
    0   94.6000  125.0000  102.0000  236.0000  406.0000  345.0000  438.0000  242.0000
  94.6000       0  154.0000  187.0000  333.0000  480.0000  250.0000  519.0000  327.0000
 125.0000  154.0000       0  204.0000  228.0000  444.0000  337.0000  335.0000  287.0000
 102.0000  187.0000  204.0000       0  145.0000  277.0000  352.0000  409.0000  200.0000
 236.0000  333.0000  228.0000  145.0000       0  122.0000  499.0000  352.0000  157.0000
 406.0000  480.0000  444.0000  277.0000  122.0000       0  707.0000  384.0000  191.0000
 345.0000  250.0000  337.0000  352.0000  499.0000  707.0000       0  764.0000  574.0000
 438.0000  519.0000  335.0000  409.0000  352.0000  384.0000  764.0000       0  193.0000
 242.0000  327.0000  287.0000  200.0000  157.0000  191.0000  574.0000  193.0000       0
```

D'après les tests paramétriques, on a décidé d'utilisé les paramethre suivant :

Taille de la population 50

Nombre d'itération 100

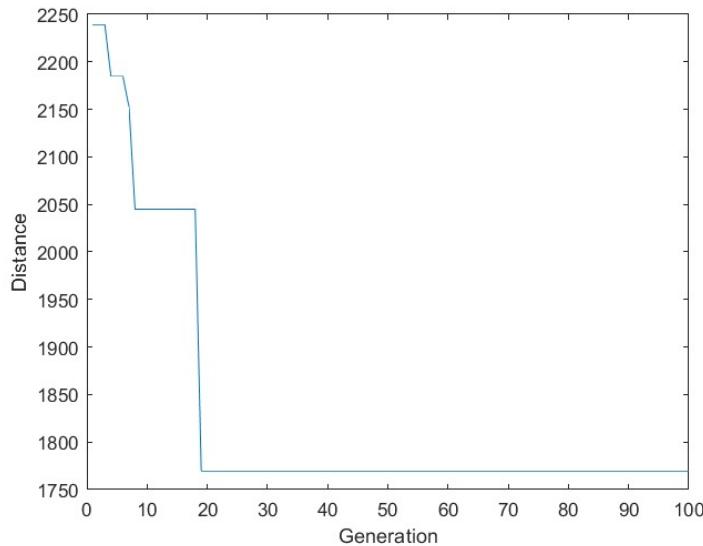
Probabilité d'utiliser la roulette comme méthode de sélection : $P_R = 1$

Probabilité d'utiliser le croisement par la méthode OX : $ox=1$

Probabilité d'utiliser le facteur de mutation sur un individu : $Pm=0.05$

Taille de l'ilitisme 10

On obtient le résultat suivant durant les itérations :



La solution optimale est :

chemin : 2 1 4 5 6 9 8 3 7

distance : 1769.6 km

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 problème on l'a traité avec l'algorithme génétique l'un des méthodes les plus utilisées dans ce domaine d'optimisation. et on a constaté que cette méthode est très efficace dans la vie quotidienne. Pour cela on est ambitieux de bien la découvrir de plus proche et voir toutes les combinaisons possibles de cet algorithme avec les autres algorithmes. Enfin l'appliquer sur beaucoup d'autres problèmes.

Chapitre 5

Annexe

main tsp function

```
1
2     function [pop , Best_Distances , Best_Chemin s ,
3         dist_moyene ]=Main( Cout_mat , n_p , n_i , Ps , P_R , ox ,
4         Taille , Pm , il )
5         % declarer x global pour que l'on peut y' acc der
6         % depuis la fitness
7         %global X;
8         % V un table contenet le nom des villes est ses
9         % cordonn s
10        y = size(Cout_mat);
11        n_v = y(1);
12        pop = pop_gen(n_p , n_v );
13        n_p_S = floor(n_p*Ps);
14
15        Best_Chemin s = zeros(n_i ,n_v );      %Le vecteur
16        % dans lequel on stock la meilleure valeur pour
17        % chaque iteration
18        Best_Distances = zeros(1 ,n_i );
19        dist_moyene = zeros(1 ,n_i );
20
21        im = il ;
22        for i=1:n_i
23            [fitness , Couts]=Fitness (Cout_mat ,pop );
24            [best_distance , best_chemin] = best_Cout(Couts ,
25                pop );
26            Best_Chemin s(i ,:) = best_chemin ;
27            Best_Distances(i) = best_distance ;
28            dist_moyene(i) = mean(Couts );
29
30            Parents = Selection(fitness ,Couts ,pop , Taille ,
31                n_p_S ,P_R );
32            Enfants = croisement(Parents ,ox );
```

```

25 Enfants = Mutation(Enfants ,Pm) ;
26 pop = insertion(Enfants ,pop ,Couts) ;
27 if im == i
28 pop = max_illemination(pop ,Cout_mat) ;
29 im = im + il ;
30 end
31 disp(['Generation : ',num2str(i)]);
32 disp(['Meilleur Trajet : ',num2str(best_chemin)])
33 ;
34 disp(['Distance : ', num2str(best_distance)]);
35 end
36 figure
37 plot([1:n_i],Best_Distances ,[1:n_i],dist_moyene);
38 xlabel('Generation');
39 ylabel('Distance');
40 [best,k] = min(Best_Distances);
41 disp(['Meilleur Trajet Trouv : ',num2str(
42 Best_Chemins(k,:))]);
43 disp(['Distance : ',num2str(best)]);
44 end

```

Fonction fitness permet d evaluer la population

```

1 function [ fitness , Couts]=Fitness (Cout_mat,pop)
2
3 %nombre de population et nombre des villes :
4 [n_p , n_v] = size(pop);
5
6 % initialisation d'u vecteur couts avec des zeros
7 Couts = zeros(1 , n_p);
8
9 %procedure des calcules des couts
10
11 for i=1:n_p
12 S = pop(i ,:);
13 for j = 1: (n_v - 1)
14 Couts(i) = Couts(i) + Cout_mat(S(j),S(j+1));
15 end
16 Couts(i) = Couts(i) + Cout_mat(S(n_v),S(1));
17 end
18
19 M = max(Couts);
20 fitness = M - Couts;
21 end
22
23 function [ best_distance , best_chemin] = best_Cout
24 (Couts,pop)

```

```

2 [ best_distance , i ] = min(Couts);
3 best_chemin = pop(i,:);
4 end

        SELECTION

1
2 function [PARENTS] = Selection(fitness,Couts,pop,
3     Taille,n_p_S,P_R)
4 r = rand;
5 %P_R = 1-P_T = proba d'utiliser la roulette dans
6 %chaque iteration

7 if r <=P_R
8 PARENTS = Roulette_selection(fitness,pop,n_p_S);
9 else
10 PARENTS = TOURNOI(Couts,pop,Taille,n_p_S);
11 end
11 end

        Roulette

1
2 function [pop_S] = Roulette_selection(fitness,pop
3 ,n_p_S)
4
5 %pop : premier population
6 %n_p : taille population
7 %n_v ;: Nombre des villes

8 %enregistres les information
9 [~,n_v] = size(pop);

10 %n_p_s : Nombre du population select
11
12
13
14 %pop_S : population select
15 pop_S = zeros(n_p_S,n_v);

16
17 %cree un vector des fitness accumel : fit_acc
18 fit_acc = zeros(1,length(fitness));
19 s = 0;

20
21 for j=1:length(fitness)
22 fit_acc(j) = s + fitness(j);
23 s = fit_acc(j);
24 end

25
26

```

```

27
28     % procedure de selection
29     for i=1:n_p_S
30
31         %genere un nombre aleatoire entre [min(fit_acc);
32         %                                     max(fit_acc)]
33         r = (max(fit_acc)-min(fit_acc))*rand() + min(
34             fit_acc);
35
36         %prendre la premiere valeur fit_acc plus grande
37         % que r
38
39         for j=1:length(fit_acc)
40
41             if r <= fit_acc(j)
42                 pop_S(i,:) = pop(j,:);
43                 break;
44             end
45         end
46     end
47 
```

Tournoi

```

1
2     function [ Parents ] = TOURNOI(Couts ,pop , Taille ,
3         n_p_S)
4
5         %selection de la population qui fait le tournoi
6         % calculer le nombre total des gagnants
7         % Taille = la taille de chaque tournoi
8
9         [~,n_v] = size(pop);
10        Parents = zeros(n_p_S,n_v);
11        Winners = zeros(n_p_S,1);
12
13        % grand boucle sur tout la population selectionne
14        for j = 1:n_p_S
15            T = [];
16            % choisir les candidats
17            for i = 1:Taille
18                r = randi(length(Couts));
19                while ismember(r,T)==1
20                    r = randi(length(Couts));
21                end
22                T = [T,r];
23            end
24
25            winner = Couts(T(1));
26            c = T(1);
27 
```

```

23    % depart du tournemet
24    for i = 2: Taille
25    if Couts(T(i)) <= winner
26        winner = Couts(T(i));
27        c = T(i) ;
28    end
29
30    end
31    Winners(j) = c;
32    % Winners un vecteur content les indice des
33        gangnats
34    end
35
36    for i = 1 : n_p_S
37        Parents(i,:) = pop(Winners(i),:);
38    end
39

```

Croisement

```

1
2    function [Enfants]=croisement(pop,ox)
3    X = size(pop);
4    %T : table d indice
5    T = [];
6
7    Enfants = [];
8    %boocles sur la moiti du population
9    for n = 1:floor(X(1)/2)
10        %choisir l idice de pere
11        i = randi(X(1));
12        %eviter la repetition des pere
13        while ismember(i,T)==1
14            i = randi(X(1));
15        end
16
17        T = [T; i];
18        % meme procedure pour la mere
19        j = randi(X(1));
20        while ismember(j ,T)==1
21            j = randi(X(1));
22        end
23        T = [T; j];
24
25        %choisir aleatoirement le type de croisement a
26        appliqu

```

```

26      r = rand ;
27      % croisement OX
28      if r <= ox
29          [enfant1 , enfant2 ]=do_ox(pop(i ,:) ,pop(j ,:) );
30
31      %croisement PMX
32      else
33          [enfant1 , enfant2 ] = pmx(pop(i ,:) ,pop(j ,:) );
34      end
35      Enfants = [Enfants ;[enfant1 ;enfant2 ]];
36      end
37      end

```

Croisement OX

```

1
2      function [enfant1 , enfant2 ]=do_ox(pere ,mere)
3
4          n_v=length(pere);
5
6          c = randsample(n_v-1,2); % Retourner deux point
7              de courpures
8          point1=min(c);
9          point2=max(c);
10
11         enfant1=pere(point1+1:point2);
12         [~, entre_courpure] = ismember(enfant1 ,mere);
13         entre_courpure=sort(entre_courpure);
14         enfant1=[pere(1:point1) ,mere(entre_courpure) ,
15                 pere(point2+1:end)];
16
17         enfant2=mere(point1+1:point2);
18         [~, entre_courpure1]=ismember(enfant2 ,pere);
19         entre_courpure1=sort(entre_courpure1);
20         enfant2=[mere(1:point1) , pere(entre_courpure1) ,
21                 mere(point2+1:end)];
22
23     end

```

Croisement PMX

```

1
2      function [ child1 , child2 ] = pmx(parent1 ,
3          parent2)
4          copure1 = randi(length(parent1));
5          copure2 = randi(length(parent1));
6          while copure2 == copure1
7              copure2 = randi(length(parent2));

```

```

7      end
8
9      child1 = zeros(size(parent1));
10     child2 = zeros(size(parent2));
11     %taille de copure
12     % premiere phase du croisement bpmx
13     child1(min(copure1,copure2)+1:max(copure1,copure2)
14         ) = parent2(min(copure1,copure2)+1:max(
15             copure1,copure2));
16     child2(min(copure1,copure2)+1:max(copure1,copure2)
17         ) = parent1(min(copure1,copure2)+1:max(
18             copure1,copure2));
19     missing1 = [];
20     missing2 = [];
21
22     %deuxieme
23     for i = 1:length(parent1)
24         %table existence d un element du parent dans o o
25             qui designe enfant
26         txo1 = ismember(parent1(i),child1);
27         txo2 = ismember(parent2(i), child2);
28         txo3 = ismember(parent2(i),child1);
29         txo4 = ismember(parent1(i),child2);
30         if txo2 == 0
31
32             %on s assure que seul les valeur qui on zero
33             sont modier
34             %one peut faire de meme change en changeant les
35             autre de la
36             %boucle
37             if (child2(i) == 0)
38                 child2(i)=parent2(i);
39             end
40             else
41                 if txo3 ==0
42                     missing2 = [ missing2 ; [ i , parent2(i) ]];
43                 end
44             end
45
46             if txo1 == 0
47                 if (child1(i)==0)
48                     child1(i)=parent1(i);
49                 end
50                 else
51                     if txo4 ==0
52                         missing1 = [ missing1 ; [ i , parent1(i) ]];
53                     end
54                 end
55             end
56         end
57     end
58
59     if txo1 == 0
60         if (child1(i)==0)
61             child1(i)=parent1(i);
62         end
63         else
64             if txo4 ==0
65                 missing1 = [ missing1 ; [ i , parent1(i) ]];
66             end
67         end
68     end
69
70     if txo2 == 0
71         if (child2(i)==0)
72             child2(i)=parent2(i);
73         end
74         else
75             if txo3 ==0
76                 missing2 = [ missing2 ; [ i , parent2(i) ]];
77             end
78         end
79     end
80
81     if txo3 == 0
82         if (child1(i)==0)
83             child1(i)=parent1(i);
84         end
85         else
86             if txo4 ==0
87                 missing1 = [ missing1 ; [ i , parent1(i) ]];
88             end
89         end
90     end
91
92     if txo4 == 0
93         if (child2(i)==0)
94             child2(i)=parent2(i);
95         end
96         else
97             if txo3 ==0
98                 missing2 = [ missing2 ; [ i , parent2(i) ]];
99             end
100            end
101        end
102    end
103
104    if txo1 == 0
105        if (child1(i)==0)
106            child1(i)=parent1(i);
107        end
108        else
109            if txo4 ==0
110                missing1 = [ missing1 ; [ i , parent1(i) ]];
111            end
112        end
113    end
114
115    if txo2 == 0
116        if (child2(i)==0)
117            child2(i)=parent2(i);
118        end
119        else
120            if txo3 ==0
121                missing2 = [ missing2 ; [ i , parent2(i) ]];
122            end
123        end
124    end
125
126    if txo3 == 0
127        if (child1(i)==0)
128            child1(i)=parent1(i);
129        end
130        else
131            if txo4 ==0
132                missing1 = [ missing1 ; [ i , parent1(i) ]];
133            end
134        end
135    end
136
137    if txo4 == 0
138        if (child2(i)==0)
139            child2(i)=parent2(i);
140        end
141        else
142            if txo3 ==0
143                missing2 = [ missing2 ; [ i , parent2(i) ]];
144            end
145        end
146    end
147
148    if txo1 == 0
149        if (child1(i)==0)
150            child1(i)=parent1(i);
151        end
152        else
153            if txo4 ==0
154                missing1 = [ missing1 ; [ i , parent1(i) ]];
155            end
156        end
157    end
158
159    if txo2 == 0
160        if (child2(i)==0)
161            child2(i)=parent2(i);
162        end
163        else
164            if txo3 ==0
165                missing2 = [ missing2 ; [ i , parent2(i) ]];
166            end
167        end
168    end
169
170    if txo3 == 0
171        if (child1(i)==0)
172            child1(i)=parent1(i);
173        end
174        else
175            if txo4 ==0
176                missing1 = [ missing1 ; [ i , parent1(i) ]];
177            end
178        end
179    end
180
181    if txo4 == 0
182        if (child2(i)==0)
183            child2(i)=parent2(i);
184        end
185        else
186            if txo3 ==0
187                missing2 = [ missing2 ; [ i , parent2(i) ]];
188            end
189        end
190    end
191
192    if txo1 == 0
193        if (child1(i)==0)
194            child1(i)=parent1(i);
195        end
196        else
197            if txo4 ==0
198                missing1 = [ missing1 ; [ i , parent1(i) ]];
199            end
200        end
201    end
202
203    if txo2 == 0
204        if (child2(i)==0)
205            child2(i)=parent2(i);
206        end
207        else
208            if txo3 ==0
209                missing2 = [ missing2 ; [ i , parent2(i) ]];
210            end
211        end
212    end
213
214    if txo3 == 0
215        if (child1(i)==0)
216            child1(i)=parent1(i);
217        end
218        else
219            if txo4 ==0
220                missing1 = [ missing1 ; [ i , parent1(i) ]];
221            end
222        end
223    end
224
225    if txo4 == 0
226        if (child2(i)==0)
227            child2(i)=parent2(i);
228        end
229        else
230            if txo3 ==0
231                missing2 = [ missing2 ; [ i , parent2(i) ]];
232            end
233        end
234    end
235
236    if txo1 == 0
237        if (child1(i)==0)
238            child1(i)=parent1(i);
239        end
240        else
241            if txo4 ==0
242                missing1 = [ missing1 ; [ i , parent1(i) ]];
243            end
244        end
245    end
246
247    if txo2 == 0
248        if (child2(i)==0)
249            child2(i)=parent2(i);
250        end
251        else
252            if txo3 ==0
253                missing2 = [ missing2 ; [ i , parent2(i) ]];
254            end
255        end
256    end
257
258    if txo3 == 0
259        if (child1(i)==0)
260            child1(i)=parent1(i);
261        end
262        else
263            if txo4 ==0
264                missing1 = [ missing1 ; [ i , parent1(i) ]];
265            end
266        end
267    end
268
269    if txo4 == 0
270        if (child2(i)==0)
271            child2(i)=parent2(i);
272        end
273        else
274            if txo3 ==0
275                missing2 = [ missing2 ; [ i , parent2(i) ]];
276            end
277        end
278    end
279
280    if txo1 == 0
281        if (child1(i)==0)
282            child1(i)=parent1(i);
283        end
284        else
285            if txo4 ==0
286                missing1 = [ missing1 ; [ i , parent1(i) ]];
287            end
288        end
289    end
290
291    if txo2 == 0
292        if (child2(i)==0)
293            child2(i)=parent2(i);
294        end
295        else
296            if txo3 ==0
297                missing2 = [ missing2 ; [ i , parent2(i) ]];
298            end
299        end
300    end
301
302    if txo3 == 0
303        if (child1(i)==0)
304            child1(i)=parent1(i);
305        end
306        else
307            if txo4 ==0
308                missing1 = [ missing1 ; [ i , parent1(i) ]];
309            end
310        end
311    end
312
313    if txo4 == 0
314        if (child2(i)==0)
315            child2(i)=parent2(i);
316        end
317        else
318            if txo3 ==0
319                missing2 = [ missing2 ; [ i , parent2(i) ]];
320            end
321        end
322    end
323
324    if txo1 == 0
325        if (child1(i)==0)
326            child1(i)=parent1(i);
327        end
328        else
329            if txo4 ==0
330                missing1 = [ missing1 ; [ i , parent1(i) ]];
331            end
332        end
333    end
334
335    if txo2 == 0
336        if (child2(i)==0)
337            child2(i)=parent2(i);
338        end
339        else
340            if txo3 ==0
341                missing2 = [ missing2 ; [ i , parent2(i) ]];
342            end
343        end
344    end
345
346    if txo3 == 0
347        if (child1(i)==0)
348            child1(i)=parent1(i);
349        end
350        else
351            if txo4 ==0
352                missing1 = [ missing1 ; [ i , parent1(i) ]];
353            end
354        end
355    end
356
357    if txo4 == 0
358        if (child2(i)==0)
359            child2(i)=parent2(i);
360        end
361        else
362            if txo3 ==0
363                missing2 = [ missing2 ; [ i , parent2(i) ]];
364            end
365        end
366    end
367
368    if txo1 == 0
369        if (child1(i)==0)
370            child1(i)=parent1(i);
371        end
372        else
373            if txo4 ==0
374                missing1 = [ missing1 ; [ i , parent1(i) ]];
375            end
376        end
377    end
378
379    if txo2 == 0
380        if (child2(i)==0)
381            child2(i)=parent2(i);
382        end
383        else
384            if txo3 ==0
385                missing2 = [ missing2 ; [ i , parent2(i) ]];
386            end
387        end
388    end
389
390    if txo3 == 0
391        if (child1(i)==0)
392            child1(i)=parent1(i);
393        end
394        else
395            if txo4 ==0
396                missing1 = [ missing1 ; [ i , parent1(i) ]];
397            end
398        end
399    end
400
401    if txo4 == 0
402        if (child2(i)==0)
403            child2(i)=parent2(i);
404        end
405        else
406            if txo3 ==0
407                missing2 = [ missing2 ; [ i , parent2(i) ]];
408            end
409        end
410    end
411
412    if txo1 == 0
413        if (child1(i)==0)
414            child1(i)=parent1(i);
415        end
416        else
417            if txo4 ==0
418                missing1 = [ missing1 ; [ i , parent1(i) ]];
419            end
420        end
421    end
422
423    if txo2 == 0
424        if (child2(i)==0)
425            child2(i)=parent2(i);
426        end
427        else
428            if txo3 ==0
429                missing2 = [ missing2 ; [ i , parent2(i) ]];
430            end
431        end
432    end
433
434    if txo3 == 0
435        if (child1(i)==0)
436            child1(i)=parent1(i);
437        end
438        else
439            if txo4 ==0
440                missing1 = [ missing1 ; [ i , parent1(i) ]];
441            end
442        end
443    end
444
445    if txo4 == 0
446        if (child2(i)==0)
447            child2(i)=parent2(i);
448        end
449        else
450            if txo3 ==0
451                missing2 = [ missing2 ; [ i , parent2(i) ]];
452            end
453        end
454    end
455
456    if txo1 == 0
457        if (child1(i)==0)
458            child1(i)=parent1(i);
459        end
460        else
461            if txo4 ==0
462                missing1 = [ missing1 ; [ i , parent1(i) ]];
463            end
464        end
465    end
466
467    if txo2 == 0
468        if (child2(i)==0)
469            child2(i)=parent2(i);
470        end
471        else
472            if txo3 ==0
473                missing2 = [ missing2 ; [ i , parent2(i) ]];
474            end
475        end
476    end
477
478    if txo3 == 0
479        if (child1(i)==0)
480            child1(i)=parent1(i);
481        end
482        else
483            if txo4 ==0
484                missing1 = [ missing1 ; [ i , parent1(i) ]];
485            end
486        end
487    end
488
489    if txo4 == 0
490        if (child2(i)==0)
491            child2(i)=parent2(i);
492        end
493        else
494            if txo3 ==0
495                missing2 = [ missing2 ; [ i , parent2(i) ]];
496            end
497        end
498    end
499
500    if txo1 == 0
501        if (child1(i)==0)
502            child1(i)=parent1(i);
503        end
504        else
505            if txo4 ==0
506                missing1 = [ missing1 ; [ i , parent1(i) ]];
507            end
508        end
509    end
510
511    if txo2 == 0
512        if (child2(i)==0)
513            child2(i)=parent2(i);
514        end
515        else
516            if txo3 ==0
517                missing2 = [ missing2 ; [ i , parent2(i) ]];
518            end
519        end
520    end
521
522    if txo3 == 0
523        if (child1(i)==0)
524            child1(i)=parent1(i);
525        end
526        else
527            if txo4 ==0
528                missing1 = [ missing1 ; [ i , parent1(i) ]];
529            end
530        end
531    end
532
533    if txo4 == 0
534        if (child2(i)==0)
535            child2(i)=parent2(i);
536        end
537        else
538            if txo3 ==0
539                missing2 = [ missing2 ; [ i , parent2(i) ]];
540            end
541        end
542    end
543
544    if txo1 == 0
545        if (child1(i)==0)
546            child1(i)=parent1(i);
547        end
548        else
549            if txo4 ==0
550                missing1 = [ missing1 ; [ i , parent1(i) ]];
551            end
552        end
553    end
554
555    if txo2 == 0
556        if (child2(i)==0)
557            child2(i)=parent2(i);
558        end
559        else
560            if txo3 ==0
561                missing2 = [ missing2 ; [ i , parent2(i) ]];
562            end
563        end
564    end
565
566    if txo3 == 0
567        if (child1(i)==0)
568            child1(i)=parent1(i);
569        end
570        else
571            if txo4 ==0
572                missing1 = [ missing1 ; [ i , parent1(i) ]];
573            end
574        end
575    end
576
577    if txo4 == 0
578        if (child2(i)==0)
579            child2(i)=parent2(i);
580        end
581        else
582            if txo3 ==0
583                missing2 = [ missing2 ; [ i , parent2(i) ]];
584            end
585        end
586    end
587
588    if txo1 == 0
589        if (child1(i)==0)
590            child1(i)=parent1(i);
591        end
592        else
593            if txo4 ==0
594                missing1 = [ missing1 ; [ i , parent1(i) ]];
595            end
596        end
597    end
598
599    if txo2 == 0
600        if (child2(i)==0)
601            child2(i)=parent2(i);
602        end
603        else
604            if txo3 ==0
605                missing2 = [ missing2 ; [ i , parent2(i) ]];
606            end
607        end
608    end
609
610    if txo3 == 0
611        if (child1(i)==0)
612            child1(i)=parent1(i);
613        end
614        else
615            if txo4 ==0
616                missing1 = [ missing1 ; [ i , parent1(i) ]];
617            end
618        end
619    end
620
621    if txo4 == 0
622        if (child2(i)==0)
623            child2(i)=parent2(i);
624        end
625        else
626            if txo3 ==0
627                missing2 = [ missing2 ; [ i , parent2(i) ]];
628            end
629        end
630    end
631
632    if txo1 == 0
633        if (child1(i)==0)
634            child1(i)=parent1(i);
635        end
636        else
637            if txo4 ==0
638                missing1 = [ missing1 ; [ i , parent1(i) ]];
639            end
640        end
641    end
642
643    if txo2 == 0
644        if (child2(i)==0)
645            child2(i)=parent2(i);
646        end
647        else
648            if txo3 ==0
649                missing2 = [ missing2 ; [ i , parent2(i) ]];
650            end
651        end
652    end
653
654    if txo3 == 0
655        if (child1(i)==0)
656            child1(i)=parent1(i);
657        end
658        else
659            if txo4 ==0
660                missing1 = [ missing1 ; [ i , parent1(i) ]];
661            end
662        end
663    end
664
665    if txo4 == 0
666        if (child2(i)==0)
667            child2(i)=parent2(i);
668        end
669        else
670            if txo3 ==0
671                missing2 = [ missing2 ; [ i , parent2(i) ]];
672            end
673        end
674    end
675
676    if txo1 == 0
677        if (child1(i)==0)
678            child1(i)=parent1(i);
679        end
680        else
681            if txo4 ==0
682                missing1 = [ missing1 ; [ i , parent1(i) ]];
683            end
684        end
685    end
686
687    if txo2 == 0
688        if (child2(i)==0)
689            child2(i)=parent2(i);
690        end
691        else
692            if txo3 ==0
693                missing2 = [ missing2 ; [ i , parent2(i) ]];
694            end
695        end
696    end
697
698    if txo3 == 0
699        if (child1(i)==0)
700            child1(i)=parent1(i);
701        end
702        else
703            if txo4 ==0
704                missing1 = [ missing1 ; [ i , parent1(i) ]];
705            end
706        end
707    end
708
709    if txo4 == 0
710        if (child2(i)==0)
711            child2(i)=parent2(i);
712        end
713        else
714            if txo3 ==0
715                missing2 = [ missing2 ; [ i , parent2(i) ]];
716            end
717        end
718    end
719
720    if txo1 == 0
721        if (child1(i)==0)
722            child1(i)=parent1(i);
723        end
724        else
725            if txo4 ==0
726                missing1 = [ missing1 ; [ i , parent1(i) ]];
727            end
728        end
729    end
730
731    if txo2 == 0
732        if (child2(i)==0)
733            child2(i)=parent2(i);
734        end
735        else
736            if txo3 ==0
737                missing2 = [ missing2 ; [ i , parent2(i) ]];
738            end
739        end
740    end
741
742    if txo3 == 0
743        if (child1(i)==0)
744            child1(i)=parent1(i);
745        end
746        else
747            if txo4 ==0
748                missing1 = [ missing1 ; [ i , parent1(i) ]];
749            end
750        end
751    end
752
753    if txo4 == 0
754        if (child2(i)==0)
755            child2(i)=parent2(i);
756        end
757        else
758            if txo3 ==0
759                missing2 = [ missing2 ; [ i , parent2(i) ]];
760            end
761        end
762    end
763
764    if txo1 == 0
765        if (child1(i)==0)
766            child1(i)=parent1(i);
767        end
768        else
769            if txo4 ==0
770                missing1 = [ missing1 ; [ i , parent1(i) ]];
771            end
772        end
773    end
774
775    if txo2 == 0
776        if (child2(i)==0)
777            child2(i)=parent2(i);
778        end
779        else
780            if txo3 ==0
781                missing2 = [ missing2 ; [ i , parent2(i) ]];
782            end
783        end
784    end
785
786    if txo3 == 0
787        if (child1(i)==0)
788            child1(i)=parent1(i);
789        end
790        else
791            if txo4 ==0
792                missing1 = [ missing1 ; [ i , parent1(i) ]];
793            end
794        end
795    end
796
797    if txo4 == 0
798        if (child2(i)==0)
799            child2(i)=parent2(i);
800        end
801        else
802            if txo3 ==0
803                missing2 = [ missing2 ; [ i , parent2(i) ]];
804            end
805        end
806    end
807
808    if txo1 == 0
809        if (child1(i)==0)
810            child1(i)=parent1(i);
811        end
812        else
813            if txo4 ==0
814                missing1 = [ missing1 ; [ i , parent1(i) ]];
815            end
816        end
817    end
818
819    if txo2 == 0
820        if (child2(i)==0)
821            child2(i)=parent2(i);
822        end
823        else
824            if txo3 ==0
825                missing2 = [ missing2 ; [ i , parent2(i) ]];
826            end
827        end
828    end
829
830    if txo3 == 0
831        if (child1(i)==0)
832            child1(i)=parent1(i);
833        end
834        else
835            if txo4 ==0
836                missing1 = [ missing1 ; [ i , parent1(i) ]];
837            end
838        end
839    end
840
841    if txo4 == 0
842        if (child2(i)==0)
843            child2(i)=parent2(i);
844        end
845        else
846            if txo3 ==0
847                missing2 = [ missing2 ; [ i , parent2(i) ]];
848            end
849        end
850    end
851
852    if txo1 == 0
853        if (child1(i)==0)
854            child1(i)=parent1(i);
855        end
856        else
857            if txo4 ==0
858                missing1 = [ missing1 ; [ i , parent1(i) ]];
859            end
860        end
861    end
862
863    if txo2 == 0
864        if (child2(i)==0)
865            child2(i)=parent2(i);
866        end
867        else
868            if txo3 ==0
869                missing2 = [ missing2 ; [ i , parent2(i) ]];
870            end
871        end
872    end
873
874    if txo3 == 0
875        if (child1(i)==0)
876            child1(i)=parent1(i);
877        end
878        else
879            if txo4 ==0
880                missing1 = [ missing1 ; [ i , parent1(i) ]];
881            end
882        end
883    end
884
885    if txo4 == 0
886        if (child2(i)==0)
887            child2(i)=parent2(i);
888        end
889        else
890            if txo3 ==0
891                missing2 = [ missing2 ; [ i , parent2(i) ]];
892            end
893        end
894    end
895
896    if txo1 == 0
897        if (child1(i)==0)
898            child1(i)=parent1(i);
899        end
900        else
901            if txo4 ==0
902                missing1 = [ missing1 ; [ i , parent1(i) ]];
903            end
904        end
905    end
906
907    if txo2 == 0
908        if (child2(i)==0)
909            child2(i)=parent2(i);
910        end
911        else
912            if txo3 ==0
913                missing2 = [ missing2 ; [ i , parent2(i) ]];
914            end
915        end
916    end
917
918    if txo3 == 0
919        if (child1(i)==0)
920            child1(i)=parent1(i);
921        end
922        else
923            if txo4 ==0
924                missing1 = [ missing1 ; [ i , parent1(i) ]];
925            end
926        end
927    end
928
929    if txo4 == 0
930        if (child2(i)==0)
931            child2(i)=parent2(i);
932        end
933        else
934            if txo3 ==0
935                missing2 = [ missing2 ; [ i , parent2(i) ]];
936            end
937        end
938    end
939
940    if txo1 == 0
941        if (child1(i)==0)
942            child1(i)=parent1(i);
943        end
944        else
945            if txo4 ==0
946                missing1 = [ missing1 ; [ i , parent1(i) ]];
947            end
948        end
949    end
950
951    if txo2 == 0
952        if (child2(i)==0)
953            child2(i)=parent2(i);
954        end
955        else
956            if txo3 ==0
957                missing2 = [ missing2 ; [ i , parent2(i) ]];
958            end
959        end
960    end
961
962    if txo3 == 0
963        if (child1(i)==0)
964            child1(i)=parent1(i);
965        end
966        else
967            if txo4 ==0
968                missing1 = [ missing1 ; [ i , parent1(i) ]];
969            end
970        end
971    end
972
973    if txo4 == 0
974        if (child2(i)==0)
975            child2(i)=parent2(i);
976        end
977        else
978            if txo3 ==0
979                missing2 = [ missing2 ; [ i , parent2(i) ]];
980            end
981        end
982    end
983
984    if txo1 == 0
985        if (child1(i)==0)
986            child1(i)=parent1(i);
987        end
988        else
989            if txo4 ==0
990                missing1 = [ missing1 ; [ i , parent1(i) ]];
991            end
992        end
993    end
994
995    if txo2 == 0
996        if (child2(i)==0)
997            child2(i)=parent2(i);
998        end
999        else
1000            if txo3 ==0
1001                missing2 = [ missing2 ; [ i , parent2(i) ]];
1002            end
1003        end
1004    end

```

```

46    end
47
48    end
49    end
50
51
52    [X Y]=size( missing2 );
53    for i=1:X
54        b =missing2(i ,2 );
55        a = missing1(i ,2 );
56        boola = true;
57        boolb = true;
58        while boolb
59
60            for j=min(copure1 ,copure2 )+1:max(copure1 ,copure2 )
61                if b==child2(j)
62                    b=child1(j);
63                    %condition d arrete
64                    boolb = true;
65                    break
66                end
67                boolb = false;
68                child2(missing2(i ,1)) = b ;
69            end
70
71        end
72        while boola
73
74            for j=min(copure1 ,copure2 )+1:max(copure1 ,copure2 )
75                if b==child1(j)
76                    b=child2(j);
77                    %condition d arrete
78                    boola = true;
79                    break
80                end
81                boola = false;
82                child1(missing1(i ,1)) = b ;
83            end
84        end
85
86    end
87    end

```

Mutation

```

1 function [ mutated_enfants ] =Mutation(enfant ,Pm)
2 [ n_p ,~ ] = size(enfant );

```

```

3     mutated_enfants = zeros( size(enfant) );
4     for i = 1:n_p
5         mutated_enfants(i,:) = do_Mutation(enfant(i,:),Pm)
6             ;
7     end
8 end

```

FUNCTION DU MUTATION POUR MUTER UN SEUL INDIVIDU

```

1
2     function [ Ind ] = do_Mutation( Ind ,Pm)
3         len = length(Ind);
4         for j = 1 : len
5             if rand <= Pm
6                 prev = Ind(j);
7                 index = randi(len);
8                 Ind(j) = Ind(index);
9                 Ind(index) = prev;
10            end
11        end
12    end

```

Insertion

```

1 % Nous compl tons la population obtenue par les
2 % Op rateurs de croisement et
3 % de Mutation, en ins rant des individus de la
4 % population initiale avec le
5 % meilleur de cette population
6
7 function pop_f = insertion(Enfants,parents,Couts)
8 [n_p_S,n_v] = size(Enfants);
9 [n_P,~] = size(parents);
10
11 pop_f = zeros(n_P,n_v);
12 % garder tous les enfants
13 pop_f(1:n_p_S,:)=Enfants;
14
15 % select aleatoirement les parents que peut
16 % servir
17 % ELITISEM
18 for i=n_E+1 : n_P-1
19     r = randi([1,n_P]);
20     pop_f(i,:)=parents(r,:);
21 end
22
23 % Garder les meilleur du parents
24 [~,i]=min(Couts);
25 pop_f(n_P,:)=parents(i,:);

```

```
23
24    end
        pour but de ileminé le plus long chemin
1
2     function [pop] = max_illemimation(pop_f, Cout_mat
3
4         )
5
6         [n_p, n_v] = size(pop_f);
7         [fitness, ~] = Fitness(Cout_mat, pop_f);
8
9         for i = 1:floor(n_p/2)
10
11             [~, j] = min(fitness);
12
13             fitness(j) = 0; %pour conserver l'ordre des
14             %chromosomes
15             end
16             pop = pop_f;
17             end
```

Références

- [1] Bourazza, S. (2006). Variantes d'algorithmes génétiques appliquées aux problèmes d'ordonnancement (Doctoral dissertation, Université du Havre).
- [2] HaJJI, O. (2003). Contribution au développement de méthodes d'optimisation stochastiques. Application à la conception des dispositifs électrotechniques. Mémoire de thèse de Doctorat, Université des sciences et technologies de Lille.
- [3] Dréo, J., Pétrowski, A., Siarry, P., & Taillard, E. (2003). Métaheuristiques pour l'optimisation difficile (p. 356). Eyrolles.
- [4] Douiri, M., Elberoussi, S., & Lakhbab, H. (2009). Cours des méthodes de résolution exactes heuristiques et métaheuristiques. Université Mohamed V, Faculté des sciences de Rabat, 5-87.
- [5] Tossa, J. (2002). Sur le Théorème d'Inversion des Spineurs de Dirac. In Annales de la Fondation Louis de Broglie (Vol. 27, No. 4, p. 597). Fondation Louis de Broglie.
- [6] Fournier, J. C. (2007). Graphes et applications. Hermès science publications-Lavoisier.