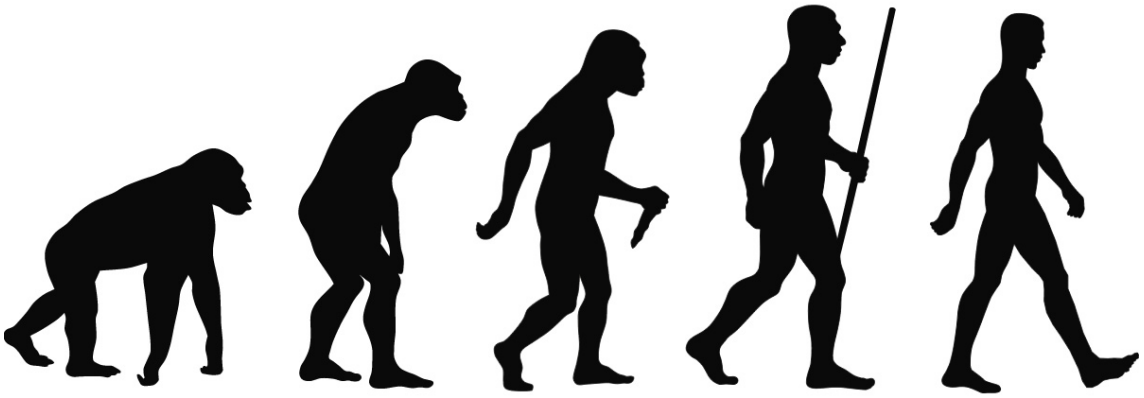
Projet semestriel : Algorithme génétique & Application au problème du voyageur de commerce



Clément COLLET Loic STEUNOU

Table des matières

1 Introduction **2**

2 Algorithme Génétique **3**
2.1 Idée et Outils nécessaires 3
2.2 Algorithme 3
2.3 Choix des deux sélections 4
2.3.1 Sélection des reproducteurs 4
2.3.2 Sélection de la population finale 5

3 Application à un problème : Le voyageur de commerce **6**
3.1 Redéfinition d'un génome et adaptation des trois fonctions outils des Algorithmes Génetiques 6
3.2 Jeux d'essais 6
3.3 Cahier des charges de l'application 8
3.3.1 Vision du projet 8
3.3.2 Fonctionnalités 9
3.3.3 Description du domaine 10

4 Implémentation **11**
4.1 Présentation des Classes 11
4.1.1 La classe Carte 11
4.1.2 La classe Genome 11
4.1.3 La classe Chemin 11
4.1.4 La classe Génération 11
4.2 Des matrices particulières 12
4.3 Premiers exemples de résultats 12
4.4 Présentation de certaines parties de code 15

5 Exploitation des résultats **20**
5.1 Comparaison des méthodes de sélection des reproducteurs 20
5.2 L'influence du nombre de génome par génération 23
5.3 L'influence du taux de mutation 28
5.4 L'importance de la conservation 29
5.5 Comparaison des méthodes de sélection de la génération suivante 30

6 Test globaux des paramètres **31**

7 Conclusion **35**

1 Introduction

Notre but pour ce projet est de comprendre le principe de l'algorithme génétique, d'en construire une modélisation UML générale et enfin de l'appliquer au problème du voyageur de commerce.

L'algorithme génétique est un exemple d'algorithme d'approximation. C'est à dire que la solution obtenue n'est pas forcément optimale mais que le coût pour l'obtenir est acceptable, tandis que l'algorithme déterministe permettant d'arriver à la solution optimale peut être trop important en termes de calcul et/ou de temps.

Le problème du voyageur de commerce est bien connu : nous disposons de n villes, toutes réliées entre elles, ainsi que le coût (en termes de distance ou de temps) pour passer de l'une à l'autre. L'objectif est de trouver le chemin qui passe par toutes les villes et revient à la ville de départ avec un coût minimal.

Pour répondre à ce problème, nous allons développer une application dans un langage orienté objet, le C++.

2 Algorithme Génétique

2.1 Idée et Outils nécessaires

Il existe deux classes d'algorithme : les algorithmes déterministes et les algorithmes d'approximations. Les premiers déterminent une solution optimale " absolue". Cependant, pour certains problèmes, une telle solution n'existe pas ou bien le coût pour l'obtenir est trop important (en termes de calcul et/ou de temps). Les algorithmes d'approximations, dont les algorithmes génétiques sont un exemple, vont nous fournir une solution approximée (ou optimale) pour un coût plus acceptable.

L'idée des algorithmes génétiques est d'avancer vers la solution, génération après génération. Une génération est un ensemble d'individus, eux même étant une "réponse" au problème donné. Un individu va être appelé génome. Quelque soit le problème que nous voulons résoudre, nous avons besoin de plusieurs outils :
- **une fonction d'Evaluation** d'un génome qui va quantifier sa réussite au problème. On cherchera à la maximiser ou bien à la minimiser en fonction du problème. On la nomme aussi fonction d'adaption ou "fitness" en anglais.
- **une fonction Reproduction** permettant le mélange de deux individus. Cette fonction doit assurer le mélange de l'information des deux génomes parents. Ce que nous définissons comme reproduction dépend bien entendu du problème et de la "nature" des génomes.
- **une fonction Mutation** qui va modifier aléatoirement une partie de l'information d'un génome, donnant ainsi naissance à une nouvelle solution. Il est important que cette mutation ne soit pas trop "radicale" afin d'assurer une stabilité au processus.
- Une fonction de **Sélection des reproducteurs** qui va nous permettre de choisir les individus se reproduisant.
- Une fonction pour la **Sélection des individus de la génération suivante**, c'est à dire ceux acceptés dans la générations suivante. Ils sont issus du groupe des reproducteurs et de celui des enfants.

Lorsque plusieurs possibilités s'offrent à nous quant à la nature d'un des outils cités plus haut, s'il y a par exemple deux manières d'évaluer la "réussite" au problème, il convient de les discuter, de les tester empiriquement afin de faire le meilleur choix. Pour l'ensemble des choix que nous aurons à faire dans nos modélisations, il n'existe que peu de résultats théoriques à ce jour, c'est pourquoi tous doivent être heuristiques.

2.2 Algorithme

Le processus est le suivant :

-**Etape 1 -** Nous générons n individus. Ces individus peuvent être le fruit du hasard ou bien être "choisis". L'avantage de partir d'individus que nous avons nous mêmes créés (que l'on a choisit tels qu'ils répondent déjà de manière acceptable au problème donné) est que l'on peut gagner du temps et éviter de considérer des solutions qui ne sont pas "intéssantes". Cependant, dans certain cas, nous n'avons pas le choix ou pas d'interêt à le faire car nous augmentons le risque de converger vers un maximum local de la fonction d'adaptation et non un maximum global. Ces individus forment notre première génération.

Début de l'itération

-**Etape 2 -** Nous choisissons les reproducteurs. Il y a différentes méthodes que nous citerons plus tard.

-**Etape 3 -** Nous générons les génomes "enfants" des reproducteurs.

-**Etape 4 -** Nous faisons muter aléatoirement les enfants. Il y a plusieurs paramètres à faire varier en fonction du problème comme la probabilité de muter pour chaque enfant etc.

-**Etape 5 -** Nous sélectionnons la génération suivante (nommons là Pk+1), c'est à dire, choisir les individus parents/enfants que nous conserverons pour la prochaine itération. Il y a différentes méthodes que nous citerons plus tard.

-**Etape 6 -** Nous regardons si un individu de Pk+1 (ou bien la génération dans son ensemble) reussît le critère d'arrêt. Il peut s'agir d'un objectif en termes de résultat à la fonction d'adaptation ou par exemple obtenir pour dix générations succesives le même meilleur génôme ou simplement en termes de temps/nombre d'itérations écoulées. On doit fixer ce critère en fonction du problème, du rapport temps d'obtention/qualité de la solution, de nos contraintes en termes de temps ou de nombre de calculs.

Si on réussit le test, on sort de l'itération sinon on retourne à l'**Etape 2 -** avec notre génération Pk+1.

Fin de l'itération

2.3 Choix des deux sélections

Comme nous l'avons dit, il y un choix à faire empiriquement pour la méthode de sélection des reproducteurs ainsi que pour la sélection des éléments de la génération Pk+1.

2.3.1 Sélection des reproducteurs

Nous voulons que plus un individu est "bon", plus il a une fonction d'adaption élévée (ou basse dans un problème de minimisation), plus il a de chance de se reproduire pour justement transmettre une part de son génome.

-La sélection par roulette

On calcule la somme S de toutes les fonctions d'adaptation d'une population. Un génome dont la fonction d'adatation renvoie k, a la probabilité k/S d'être selectionné à un tour donné. Un génome peut donc apparaître plusieurs fois dans la population des reproducteurs s'il est choisi plusieurs fois. Dans le cas d'une fonction d'adaptation que l'on chercherait à minimiser, il faudrait réadapter cette méthode.

-La sélection par rang

On trie les génomes en ordre croissant (ou décroissant si on veut minimiser la fonction d'adaptation) et on leur attribue un rang (1 pour le plus mauvais, 2 pour le 2ème moins bon etc). On fait ensuite pareil que pour la sélection par roulette, un génome ayant comme probabilité d'être choisi (son rang)/(la somme de tous les rangs).

-La sélection par tournoi

On forme aléatoirement N paires parmi nos N individus. Pour chaque paire, le génome avec la meilleure fitness a une probalité p de gagner la joute et donc de devenir reproducteur au dépens de l'autre génome. La valeur de p est encore une fois à discuter et varie en général de 70 à 100%.

-L'eugénisme

On prend tout simplement les meilleurs individus.

-La sélection uniforme

Tous les individus ont la même probabilité de devenir reproducteur.

2.3.2 Sélection de la population finale

Voici les différentes méthodes pour choisir les individus qui seront présents dans la génération Pk+1.

-Les enfants survivent

-Les meilleurs parents survivent

On se fixe un nombre k de parents qui survivent (on prend les meilleurs) et donc un nombre n-k de meilleurs enfants qui survivent aussi.

-Les meilleurs survivent

On prend les meilleurs n génomes, indépendamment du caractère parent ou enfant.

3 Application à un problème : Le voyageur de commerce

3.1 Redéfinition d'un génome et adaptation des trois fonctions outils des Algorithmes Génetiques

Dans le cas du voyageur de commerce, notre génome (individu) est une liste comprenant toutes les villes, enregistrée selon leur ordre de parcours. On va placer arbitrairement notre ville de départ comme étant la ville 0 mais cela impactera uniquement l'affichage : puisque ce parcours est une boucle, il est identique quelque soit la ville de départ.

-La fonction d'Evaluation
Nous allons simplement utiliser le coût du trajet comme variable d'évaluation, soit la somme des coûts pour passer de ville en ville. Nous sommes donc dans un cas où nous voulons miniser ce coût.

- La fonction Reproduction
Notre fonction de reproduction prend deux génomes G1 et G2 en entrée. Un entier K inférieur à N (N étant le nombre de villes) est choisi aléatoirement. Le nouveau génome fils F commence par les mêmes K villes que G1, puis nous parcourons G2 en partant du début et chaque ville qui n'est pas déjà dans F est ajoutée à la fin du génome F.

- **La fonction Mutation** Il s'agît pour nous d'une permutation de deux villes choisies aléatoirement dans un génome.

3.2 Jeux d'essais

Voici comment l'utilisateur pourra utiliser l'application :
- L'utilisateur propose à l'application un fichier texte comportant une matrice de taille n*n.
- Grâce à un fichier de données l'utilisateur peut faire varier les différents paramètres utiles à la résolution du problème du voyageur de commerce par un algorithme génétique :
- Le nombre d'individus par population
- Le nombre de générations souhaitées
- Le nombre de génomes selectionnés pour être reproducteur
- Le nombre de parents allant dans la génération suivante
- Le nombre d'enfants allant dans la génération suivante

L'application utilisera donc l'algorithme génétique pour résoudre le problème du voyageur de commerce pour la matrice proposée.
Elle retournera à l'utilisateur le meilleur génome obtenu ainsi que son coût (fitness) pour chaque génération. Elle sortira de plus, un fichier de données comportant la fitness du meilleur génome en fonction de la génération pour des analyses de résultats par la suite.

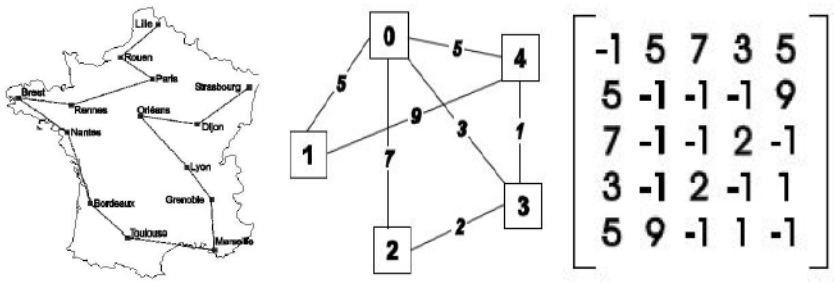


FIGURE 1 – exemple de carte représenté par un graphe et une matrice

On peut donc voir que certaines conditions sont à respecter lors du choix de la matrice et du choix de la ville de départ. Nous allons les présenter.
Contrairement à ce qui est montré dans cette matrice, nous avons modélisé le problème du voyageur de commerce avec la possibilité d'aller à n'importe quelle ville depuis n'importe quelle autre ville, il n'y a pas de valeurs (-1) en dehors de la diagonale.

Le jeu d'essai pour le nombre de villes proposées par l'utilisateur est le suivant :

Utilisateur	Application
Caractéristique du nombre proposé	**Ce que renvoi l'application**
Le nombre n'est pas entier naturel	Message d'erreur : Le nombre proposé n'est pas un entier naturel
Le nombre est entier naturel qui n'est pas compris en 1 et n	Message d'erreur : Ville de départ incorrect, entrez un entier entre 1 et n
Le nombre est un entier naturel compris en 1 et n	Ville de départ correct, l'application passe à la vérification de la matrice

FIGURE 2 – Jeu d'essai pour l'entier n représentant le nombre de villes

Le jeu d'essai pour la matrice du fichier rentrée directement dans l'application est le suivant :

Utilisateur	Application
Caractéristique de la matrice proposée	**Ce que renvoi l'application**
Le fichier ne contient pas de matrice	Message d'erreur : Matrice introuvable !
Tout les coefficients de la matrice sont sont égales à -1	Message : Il n'y a pas de chemin reliant les différentes villes
Il existe un ou plusieurs coefficients inférieur à -1	Message d'erreur : La matrice contient des valeurs non valide(< -1) !
Il existe un ou plusieurs coefficients qui ne sont pas entiers	Message d'erreur : Matrice non valide, tous les coefficients ne sont pas entier !
Il existe un ou plusieurs coefficients égaux à 0	Message d'erreur : Matrice non valide, la matrice contient des coefficients nuls !
Il existe un i tel M[i][i] = ! -1	Message d'erreur : il existe des coefficients différents de -1 sur la diagonale !
Pour tout j, il n'existe pas de i tel que M[i][j] = ! -1	Message d'erreur : il n'y a pas de chemin passant par toute les villes !
N = 1	Message : Il n'y a qu'une seule ville !
Pour toute les autres matrices n*n	Renvoi un chemin proche du chemin optimal ainsi que son coût

FIGURE 3 – Jeu d'essai pour la matrice

3.3 Cahier des charges de l'application

3.3.1 Vision du projet

Objectifs :

L'objectif de ce projet est de créer une application en C++ permettant utilisant un algorithme génétique. Nous testerons ensuite ce principe sur un problème bien connu, le problème du voyageur de commerce.

Le problème du voyageur de commerce est le suivant :

A partir d'une carte réprésentée sous la forme d'un graphe ou d'une matrice (voir Figure 1) il faut déterminer un trajet minimal permettant à un voyageur de visiter n villes en minimisant soit le temps de parcours soit la distance totale parcourue.

Oraganisation métier :

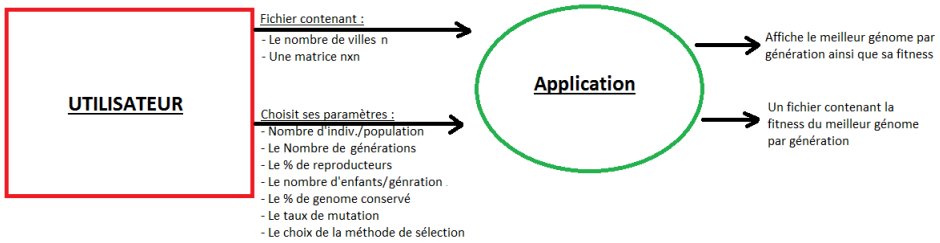


FIGURE 4 – Organisation métier

3.3.2 Fonctionnalités

Diagrammes des cas d'utilisation

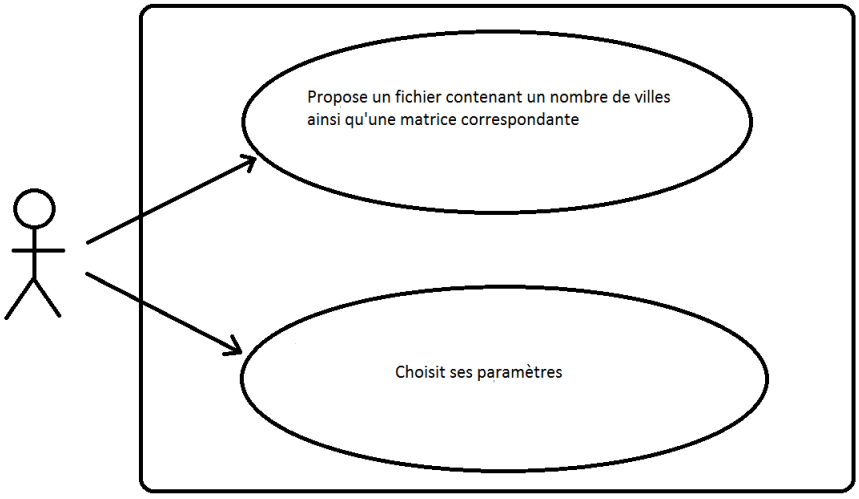


FIGURE 5 – Organisation métier

```
10
-1 1 8 9 6 11 12 8 9 6
9 -1 1 7 3 9 1 13 7 3
7 8 -1 1 7 2 8 11 13 7
1 2 3 -1 1 9 8 14 6 7
1 2 3 4 -1 1 2 3 4 12
6 7 8 9 6 -1 1 8 9 6
9 12 1 7 3 9 -1 1 7 3
7 8 7 1 7 2 8 -1 1 7
1 2 3 7 1 9 8 14 -1 1
1 2 3 4 21 10 2 3 4 -1
```

FIGURE 6 – Exemple d'un fichier entré pour une matrice de dix villes

3.3.3 Description du domaine

Diagramme de classe :

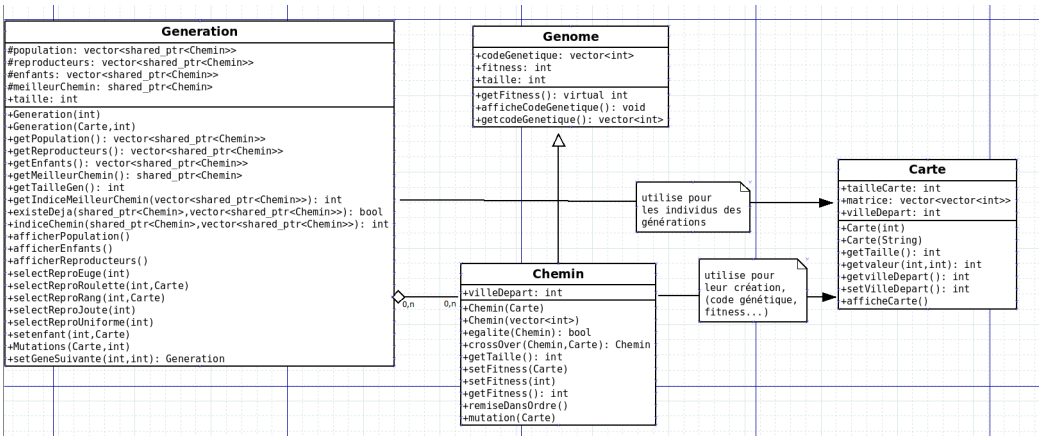


FIGURE 7 – Diagramme UML

Diagramme de séquence

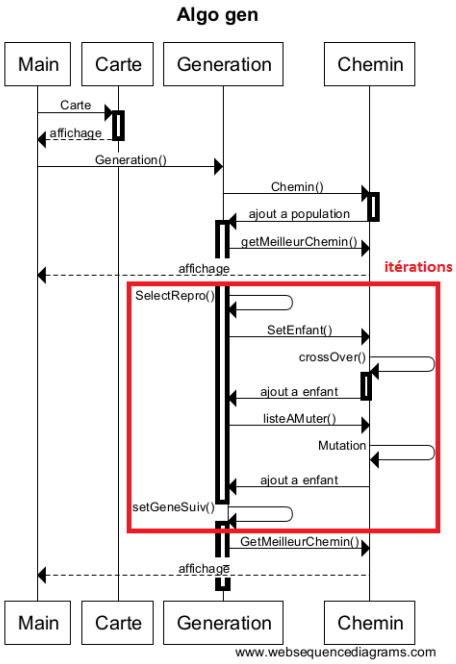


FIGURE 8 – Diagramme de séquence

4 Implémentation

Nous avons développé notre application en C++. Utiliser un langage orienté objet était adapté car nous avons pu modéliser plusieurs classes : Génome, Chemin, Génération et Carte.

Une documentation Doxygen est également disponible.

4.1 Présentation des Classes

4.1.1 La classe Carte

La classe Carte possède donc comme attribut une matrice, modélisée par un tableau dynamique d'entiers à deux dimensions. Cette matrice représente tous les coûts de passage d'une ville à une autre : en (2,4), nous trouvons le coût pour passer de la ville 2 à la ville 4. Carte possède aussi une taille, ainsi qu'une ville de départ.
L'application permet de générer des cartes aléatoires ou de les charger à partir d'un fichier. Elle dispose de différentes méthodes : getteurs et setteurs sur ces attributs ainsi que d'une méthode pour l'afficher.

4.1.2 La classe Genome

Le classe Génome est une classe abstraite de notre application, elle a pour attribut un entier correspondant à la fitness, un code génétique et un entier taille pour ce code génétique. Cette Classe pourrait théoriquement être utilisée pour tout algorithme génetique.

4.1.3 La classe Chemin

Cette classe hérite de Génome, elle représente l'application de l'algorithme génétique à notre problème. Le code génétique est une liste d'entiers correspondant aux villes et à leur ordre de visite. Nous avons utilisé "Vector" pour pouvoir avoir une taille dynamique. Elle implémente les méthodes cross-over et mutation.

4.1.4 La classe Génération

La classe Génération possède cinq attributs. Tout d'abord l'entier taille correspondant aux nombre de Chemins par géneration. Il y a ensuite trois listes de Chemins : population, reproducteurs et enfants. Nous avons choisi Vector pour nos listes, et nous utilisons des shared pointeurs sur nos Chemins. Nous expliquerons plus tard pourquoi nous avons pris des shared Pointeur et non des pointeurs classiques. Population est la liste comprenant tous les chemins de la génération actuelle. Reproducteurs est la liste des individus disponibles pour le crossover, ils sont tous présents dans Population. Enfants est la liste des enfants obtenus à la suite des crossover, de Reproducteurs et de mutation. Enfin, Géneration possède un attribut meilleurChemin qui est un shared pointeur sur le meilleur Chemin de Population. Elle possède beaucoup de méthodes qui sont expliquées dans la documentation Doxygen.

4.2 Des matrices particulières

Pour pouvoir analyser au mieux nos résultats nous avons créé des matrices de tailles 10, 25 et 50. Le point commun de ces matrices est qu'il existe pour chacune d'entre elles un chemin optimal égal à la taille de la matrice. De plus, ce chemin est :

0 1 2 3 4 5 6 7 8 9 pour la matrice de taille 10

0 1 2 3 4 5 ... 24 pour la matrice de taille 25

0 1 2 3 4 5 ... 49 pour la matrice de taille 50

Voici un exemple pour la matrice de taille 10 :

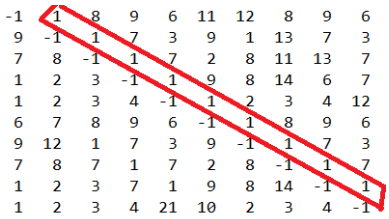


FIGURE 9 – Matrice de taille 10

On voit que pour cette matrice le chemin optimal est le chemin ayant le code génétique 0 1 2 ... 9. L'objectif de notre application sera donc de trouver un chemin ayant un code génétique avec une fitness proche ou égale à la taille de la matrice ou bien trouver le chemin optimal tout simplement. Cela pour différentes tailles de matrices (10, 25, 50).

4.3 Premiers exemples de résultats

Pour les exemples suivants, nous allons choisir des paramètres fixes pour montrer ce que l'application affiche en console.

Pour la matrice de taille 10 avec les paramètres suivants :

- Nombre de Générations maximun = 200
(Rappel : l'application s'arrête si elle trouve un génome de fitness égale à 10)

- Nombre de Génome par Génération = 300

- Méthode de séléction des reproducteurs : Eugénisme
(Voir plus haut pour les détails de cette méthode de sélection)

- Poucentage de Génomes reproducteurs = 50%
(Il y aura ici 150 individus qui seront reproducteurs, ces individus seront séléctionnés grâce à la méthode de sélection choisie ci-dessus)

- Pourcentage d'enfants = 120%
(Pourcentage de la taille de la génération, qui indique le nombre de génomes enfants qui seront crées. Exemple : taille de 100, 120% d'enfants implique que nous créons 120 enfants).

- Pourcentage d'individus conservés 5%
(On conserve les cinq meilleurs % de la population précedente dans la nouvelle génération, puis on complète avec des enfants).

- Taux de mutation = 70%
(Chaque génome enfant a 70% de chance de muter).

La console affiche le meilleur génome de la génération en cours seuleument s'il y a une amélioration de fitness par rapport à la génération précédente.

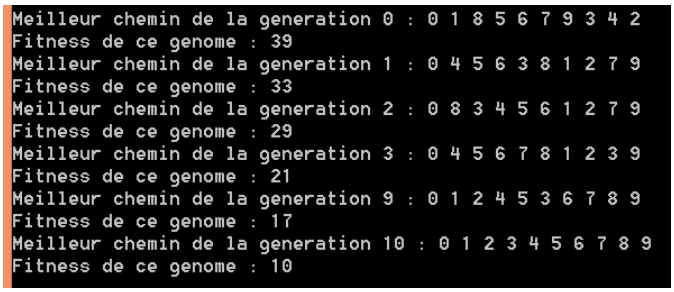


FIGURE 10 – Exemple pour une matrice de taille 10

Nous pouvons donc voir ici que l'application trouve rapidement le meilleur chemin possible pour le voyageur c'est à dire le chemin 0 1 2 ... 9 avec 10 pour fitness. Ceci peut être expliqué par le petit nombre de villes (10) ainsi que par le grand nombre de génomes par génération (300).

Pour la matrice de taille 25 :

Pour cette simulation, nous allons garder quasiment tous les paramètres égaux aux valeurs précédentes. Seul le nombre de générations va augmenter et passer de 200 à 5000 (nombre plus adapté pour cette taille de matrice). Effectivement, il y a plus de villes et donc l'application mettra plus de générations pour converger vers la solution optimale. Nous allons également augmenter le nombre d'individus par génération de 300 à 500.

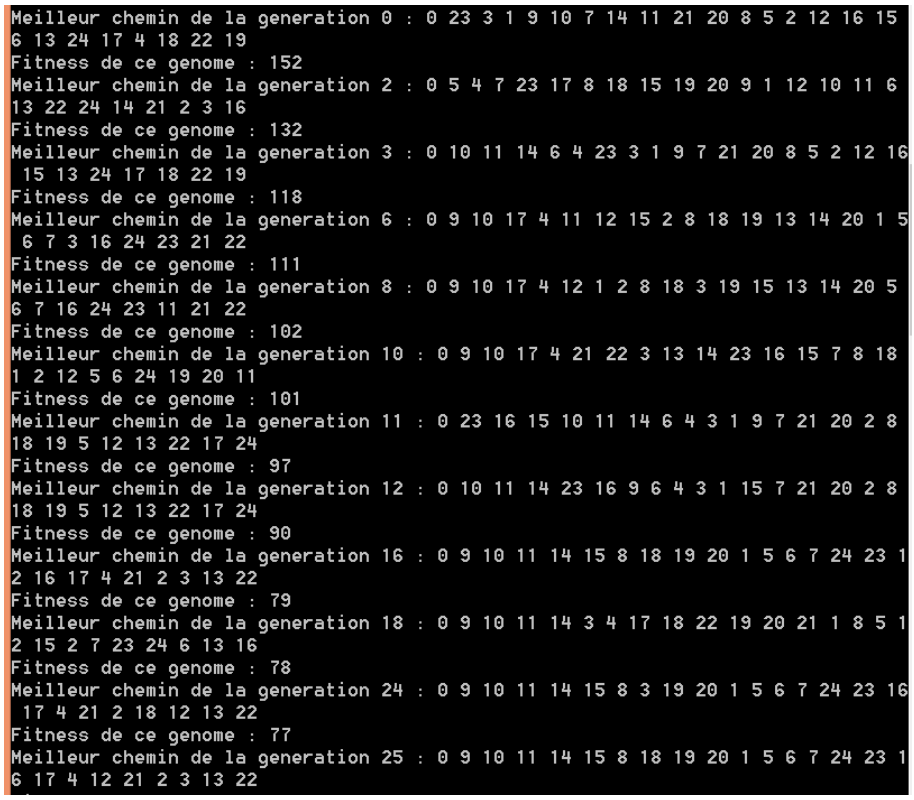


FIGURE 11 – Exemple pour une matrice de taille 25 (1)

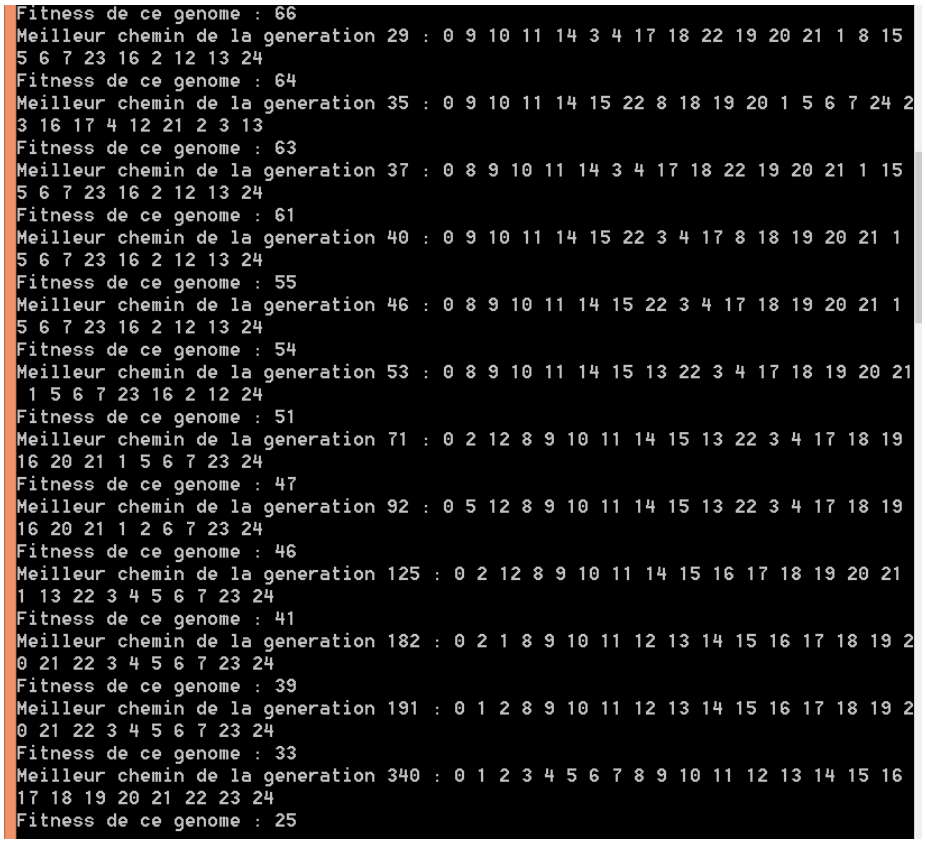


FIGURE 12 – Exemple pour une matrice de taille 25 (2)

Ici, nous pouvons voir que l'application a rapidement trouver la solution optimale. D'après la plupart de nos simulations, au bout de 5000 générations, très peu de combinaisons de paramètres trouvaient la solution optimale mais beaucoup en étaient proches. Nous pouvons donc dire que nous avons eu de la chance sur cette simulation (On peut effectivement parler de chance car les mutations sont aléatoires) mais aussi que les paramètres choisis étaient plutôt bons.

4.4 Présentation de certaines parties de code

Nous n'avons pas eu de réelles fonctions complexes dans ce projet. La principale difficulté rencontrée fut la gestion dynamique de la mémoire. En effet, nous créons et détruisons de nombreux génomes et générations. Nous devions être surs que la mémoire ne s'encombre pas, itération après itération, de données inutiles. C'est la raison pour laquel nous avons choisi d'utiliser des shared pointeurs. En effet, ils ont la particularité de supprimer l'objet pointé lorsque le dernier pointeur qui pointait dessus est détruit.

Voici la fonction qui passe de la génération K à la génération K+1 en conservant nbParents et NbEnfants.

```
1  Generation Generation::setGeneSuiv(int nbParents,int NbEnfants)
   {
3      /On initialise une generation vide de taille taille
       Generation gen = Generation(taille);
5      /On la remplit de nbParents genome qu on erase de population a
       chaque fois.
7      while(gen.population.size()<nbParents)
       {
9          int ind1 = getIndMeilleurChemin(population);
           gen.population.push_back(population.at(ind1));
11         population.erase(population.begin()+ind1);


13
       }
15     /Meme principe avec les enfants
       while(gen.population.size()<nbParents+NbEnfants)
17     {
           int ind2 = getIndMeilleurChemin(enfants);
19         gen.population.push_back(enfants.at(ind2));
           enfants.erase(enfants.begin()+ind2);
21     }
       gen.meilleurChemin = gen.population.at(getIndMeilleurChemin(
23     gen.population));
       /On finit de nettoyer la generation K
25     population.clear();
       reproducteurs.clear();
27     enfants.clear();
       return gen;
29 }
```

Voici la fonction qui choisit les reproducteurs en utilisant la méthode de la roulette. Cette méthode rend proportionnel la fitness et le pourcentage de chance d'être choisi comme reproducteur. Puisque nous voulons minimiser cette fitness, nous ne pouvons l'utiliser ainsi. Nous allons remplacer les fitness de chaque génome par la formule suivante : (la pire fitness de la génération - la fitness de ce génome + 1). Ainsi, le pire génome a une probabilité d'être choisi de 1 / (la somme des fitness) et le meilleur a la meilleure probabilité.

```
void Generation::selectRoulette(int nbRepros, Carte car)
{
    reproducteurs.clear();
    /Nous allons determiner la fitness maximum
    int maxi = 0;
    for(int i=0;i<taille;i++)
    {
        if (population.at(i)->getFitness() > maxi)
        {
            maxi = population.at(i)->getFitness();
        }
    }
    /Pour chaque genome, on modifie la fitness et on calcule en meme
    temps la somme ( s ) de toutes les fitness.
    int s=0;
    for (int i=0;i<taille;i++)
    {
        int tmp = population.at(i)->getFitness();
        int f = maxi-tmp+1;
        population.at(i)->setFitness(f);
        s+=f;
    }
    /On choisit autant de reproducteur que desirer
    while (reproducteurs.size()<nbRepros)
    {
        int it = 0;
        int i=0;
        int r = rand()%(s)+1;
        while (it<r)
        {
            it+=population.at(i)->getFitness();
            i+=1;
        }
        reproducteurs.push_back(population.at(i-1));
    }
    /On reinitialise avec la veritable fitness
    for(int i=0;i<taille;i++)
    {
        population.at(i)->setFitness(car);
    }
```

Le main

```
int main()
{

    /**< Initialisation des Parametres : */
    srand(time(0));

    /**< La carte, construction et affichage en fonction du choix
    de l utilisateur : */
    /**< On definit ici les parametres pour la carte, et on la charge
    puis on l'affiche*/
    int nombreVille = TAILLE_CARTE;
    Carte carteTest = Carte(NOM_FICHIER_ENTRE);
    carteTest.afficheCarte();
    cout<<"\n";

    /**< La premiere generation, construction et affichage du meilleur
    genome avec sa fitness */
    int tailleGen = TAILLE_GENERATION;
    Generation gen0 = Generation(carteTest,tailleGen);
    cout <<"\nMeilleur chemin de la generation 0 : ";
    gen0.getMeilleurChemin()->afficheCodeGenetique();
    cout<<"Fitness de ce genome : "
    << gen0.getMeilleurChemin()->getFitness()<<endl;

    /**< Initialisation des parametres choisis par l utilisateur
    dans le fichier de donnees */
    int nbDeBoucles=NOMBRE_DE_GENERATION;
    int nbRepro=(POURCENTAGE_INDIVIDU_PARENTS/100)*tailleGen;
    int nbPopGarde=(POURCENTAGE_INDIVIDU_CONSERVE/100)*tailleGen;
    int nbEnfants=(POURCENTAGE_ENFANTS/100)*tailleGen;

    /**< En effet le nombre d'enfants qui sera dans la generation
    suivante depend du nombre de genomes conserves ainsi
    que de la taille de la generation */
    int nbEnfantsGarde=tailleGen-nbPopGarde ;
    int choixMethode=CHOIX_DE_LA_METHODE_SELECTION_REPRODUCTEURS;
    int mutationRate = MUTATION_RATE;
    int choixNouvelleGeneration=
    CHOIX_DE_LA_METHODE_GENERATION_SUIVANTE;
```

```
48      /**< Creation du fichier qui contiendra les resultats a analyser
        (Meilleur fitness par generation) */
50      ofstream file(NOM_FICHIER_SORTIE);
        if (!file.is_open())
52      {
            cerr << "Fichier non valide";
54          exit(1);
        }
56
        /**< On effectue une boucle sur le nombre de generation souhaite,
58      si un chemin optimal est trouve (sa fitness egale a la
        taille de la carte) on s'arrete  */
60      int i=1;
        while (i<nbDeBoucles &&
62      gen0.getMeilleurChemin()->getFitness()!=nombreVille)
        {
64          /**< On stock la fitness du meilleur genome de la generation */
            int tmp = gen0.getMeilleurChemin()->getFitness();
66          /**< Premiere etape de chaque boucle, la selection des
            reproducteurs en fonction du choix de l'utilisateur  */
68          switch (choixMethode)
            {
70          case 1:
                gen0.selectReproducteurEuge(nbRepro);
72              break;
            case 2:
74              gen0.selectRoulette(nbRepro,carteTest);
                break;
76          case 3:
                gen0.selectRang(nbRepro,carteTest);
78              break;
            case 4:
80              gen0.selectReproducteurJoute(nbRepro);
                break;
82          case 5:
                gen0.selectUniforme(nbRepro);
84              break;
            }
86          /**< deuxieme etape, la creation des enfants */
            gen0.setEnfants1(nbEnfants,carteTest);
88
            /**< Troisieme etape, la mutation des enfants */
90          gen0.mutations(carteTest,mutationRate);


92


94
```

```
96          /**< Derniere etape, creation d une nouvelle generation */
            switch (choixNouvelleGeneration)
98          {
            case 1:
100             gen0 = gen0.setGeneSuiv(nbPopGarde,nbEnfantsGarde);
                break;
102         case 2:
                gen0 = gen0.setGeneSuiv();
104             break;
            }
106         /**<Si il y a une amelioration d une generation a l autre
            on affiche le meilleur genome ainsi que sa fitness */
108         if(gen0.getMeilleurChemin()->getFitness()<tmp
            || i == nbDeBoucles-1)
110         {
                //cout<<"\n";
112             cout<<"Meilleur chemin de la generation " <<i <<" : ";
                gen0.getMeilleurChemin()->afficheCodeGenetique();
114             cout<<"Fitness de ce genome : "
                << gen0.getMeilleurChemin()->getFitness()<<endl;
116         }

118         /**<On enregistre meilleure fitness de chaque
            generation dans un fichier */
120         file << i << " " << gen0.getMeilleurChemin()->getFitness();
            file << "\n";
122         i+=1;


124
        }
126     cout<<"\n";
        file.close();
128     return 0;
    }
```

5 Exploitation des résultats

Dans les prochains paragraphes, nous allons utiliser le fichier que l'application nous donne en sortie pour pouvoir travailler sur des courbes, principalement pour étudier l'évolution de la fitness du meilleur génome de chaque population au fur et à mesure que les générations évoluent.
Nous allons faire varier un à un les paramètres d'entrée :
- le nombre de génome par générations
- le nombre de reproducteurs choisis
- le nombre de parents et d'enfants choisis pour former la génération suivante
- la méthode de sélection des reproducteurs
- le taux de mutation
- la méthode de construction de la génération suivante

5.1 Comparaison des méthodes de sélection des reproducteurs

Nous allons maintenant comparer l'efficacité des différentes méthodes de sélection des reproducteurs.
Les cinq méthodes que nous avons comparées sont :
- L'eugénisme
- La sélection par roulette
- La sélection par rang
- La sélection uniforme
- Le sélection par tournoi

Nous avons donc fixé les autres paramètres. Cette façon de procéder n'est pas optimale car une fonction de sélection donnée peut être meilleure avec d'autres paramètres. Cependant, elle donne une idée de l'efficacité des méthodes. De plus, ces paramètres se sont avérés performants lors de nos différentes simulations.
Nous avons donc choisi les paramètres suivants :

Pour la matrice de taille 10 :

Nous avons décidé de ne pas faire d'étude pour cette taille de matrice car pour la quasi totalité des paramètres (sauf cas extrêmes présentés plus tard) les méthodes convergeaient assez rapidement vers la valeur optimale (la taille de la matrice 10).

Pour la matrice de taille 25 :

Prenons les paramètres suivants :
- Nombre de génome par Générations = 500
- Taux de mutation = 70%
- Nombre de Génération maximum = 5000
- Pourcentage de reproducteurs = 60%
- Pourcentage d'enfants = 120%
- Pourcentage de conservation = 5%

Sur ces courbes, la courbe rouge/orange correspond à la méthode de sélection par eugénisme, la verte la sélection par roulette, la bleue foncée par rang, la bordeaux pour la sélection uniforme et enfin la courbe jaune correspond à la sélection par tournoi.

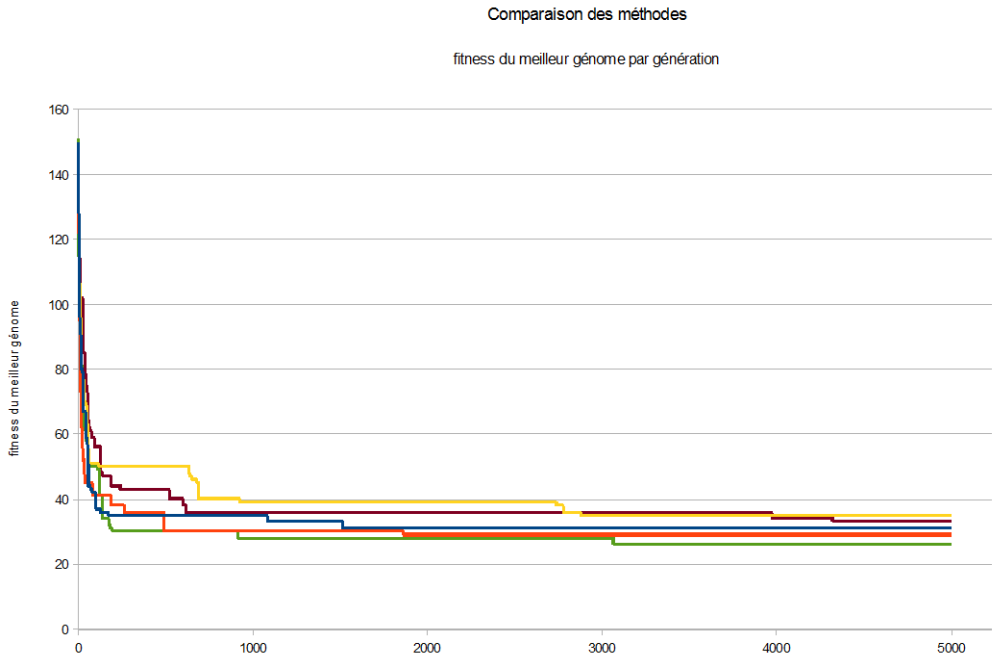


FIGURE 13 – Efficacité des méthodes de sélection (Matrice de taille 25)
axe des abscisses : numéro de la génération
axe des ordonnées : fitness du meilleur génome de cette génération

Comme il y a une conservation des meilleurs génomes d'une population à l'autre, la fitness du meilleur génome ne peut que décroitre.

On peut voir que les méthodes font converger la fitness du meilleur génome vers la valeur de 25. Cependant aucune méthode ne permet d'atteindre la valeur optimale pour ces simulations. Nous avons vu précédemment qu'avec certains paramètres et avec la méthode de sélection par eugénisme, l'application a fait converger la fitness du meilleur génome à 25 en moins de 400 générations. Cette variation dans les résultats peut s'expliquer par plusieurs raisons. Tout d'abord, 25 villes représentent un grand nombre de chemins possibles (25 !). De plus, il y a également une part de chance, en effet, les mutations étant aléatoires, il est possible qu'une mutation faisant diminuer la fitness arrive plus tôt dans certaines simulations.

Pour toute ces méthodes, le comportement des courbes est le même.La méthode de sélection par le tournoi donne un résultat un peu meilleur que les autres et la méthode de sélection du rang un résultat un peu moins bon. On ne peut donc pas donner de conclusion quant à l'efficacité des méthodes pour une matrice de taille 25. Nous avons donc fait la même expérience avec une matrice de taille 50.

En haut à droite de l'image, il est possible de voir les temps de simulation pour chaque méthode, il s'agit du dernier chiffre, celui qui précède s (pour secondes).

Pour la matrice de taille 50 :

Prenons les paramètres suivants :
- Nombre de génomes par Génération = 500
- Taux de mutation = 70%
- Nombre de Générations maximum = 15000
- Pourcentage de reproducteurs = 60%
- Pourcentage d'enfants = 120%
- Pourcentage de conservation = 5%

Sur ces courbes, la courbe rouge/orange correspond à la méthode de sélection par eugénisme, la verte la sélection par roulette, la bleue foncée par rang, la bordeaux par sélection uniforme et enfin la courbe jaune correspond à la sélection par tournoi.

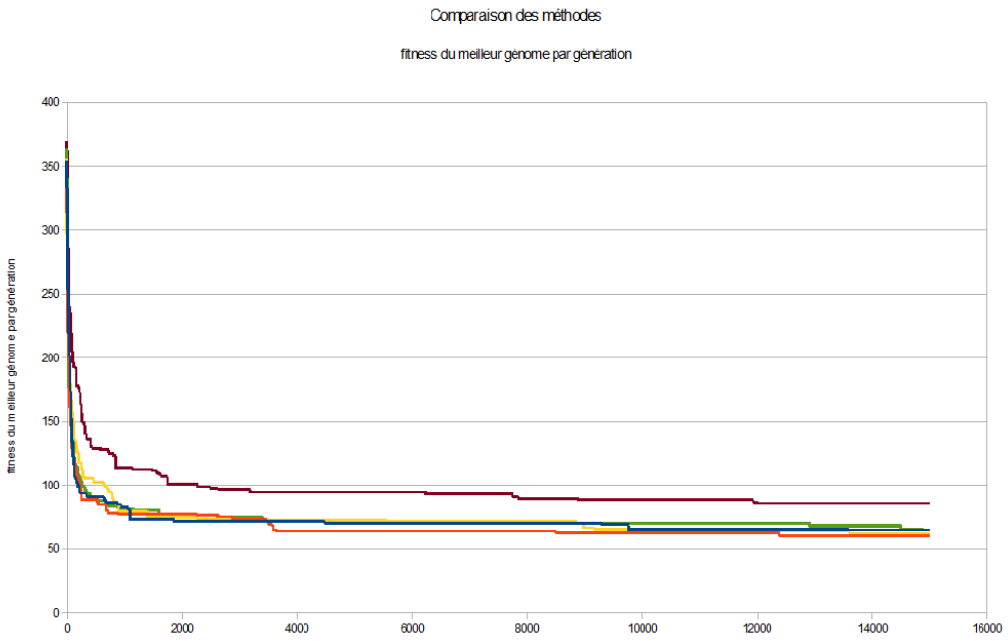


FIGURE 14 – Efficacité des méthodes de sélection (Matrice de taille 50)
axe des abscisses : numéro de la génération
axe des ordonnées : fitness du meilleur génome de cette génération

Pour la matrice de taille 50, et pour 15 000 générations, nous pouvons voir qu'aucune méthode de sélection ne permet au meilleur génome d'atteindre une fitness de 50 lors de ces simulations. Pour ces cinq méthodes, nous pouvons voir que la fitness du meilleur génome s'en rapproche, mise à part la méthode de sélection uniforme qui est moins bonne. Voici les valeurs atteintes pour les différentes méthodes de sélection des reproducteurs :

- Eugénisme : La fitness du meilleur génome en 15 000 générations maximum est 64, valeur atteinte au bout de 13566 générations

- Roulette : La fitness du meilleur génome en 15 000 générations maximum est 60, valeur atteinte au bout de 12399 générations

- Rang : La fitness du meilleur génome en 15 000 générations maximum est 61, valeur atteinte au bout de 13608 générations

- Tournoi : La fitness du meilleur génome en 15 000 générations est 64, valeur atteinte au bout de 14882 générations

- Uniforme : La fitness du meilleur génome en 15 000 générations maximum est 85, valeur atteinte au bout de 12004 générations

Nous pouvons donc considérer que la méthode de sélection uniforme (tous les individus d'une génération ont la même chance de devenir reproducteur) est moins efficace que les autres, c'était également le cas pour la quasi la totalité de nos simulations. En effet, La méthode de sélection uniforme est la méthode qui donne le plus de chance aux individus les "moins bons" de devenir reproducteur.

5.2 L'influence du nombre de génome par génération

Nous allons, dans cette partie, faire varier le nombre de génomes par génération en gardant les autres paramètres fixes. Nous allons comparer les différentes valeurs suivantes :
- 500 génomes par génération
- 200 génomes par génération
- 100 génomes par génération

Et cela pour les trois tailles de matrices 10, 25 et 50. Pour la matrice de taille 10, nous prendrons un nombre de générations maximum égal à 300, 5000 pour la matrice de taille 25 et 15000 pour la matrice de taille 50.

Nous avons choisis de fixer les autres paramètres avec des valeurs efficaces aux vues des différentes simulations que nous avons pues faire, c'est à dire aux valeurs suivantes pour les trois matrices :
- Taux de mutation = 70%
- Pourcentage de reproducteurs = 60%
- Pourcentage d'enfants = 120%
- Méthode de sélection des reproducteurs : Eugénisme
- Pourcentage de conservation = 5%

Sur les trois graphiques qui vont suivre, la courbe bleue correspond à l'évolution de la fitness du meilleur génome en fonction de la génération pour une valeur de 500 génomes par génération, la courbe verte à une valeur de 200 génomes par génération et enfin la courbe bleue à une valeur de 100 génomes pas génération.

Pour la matrice de taille 10 :

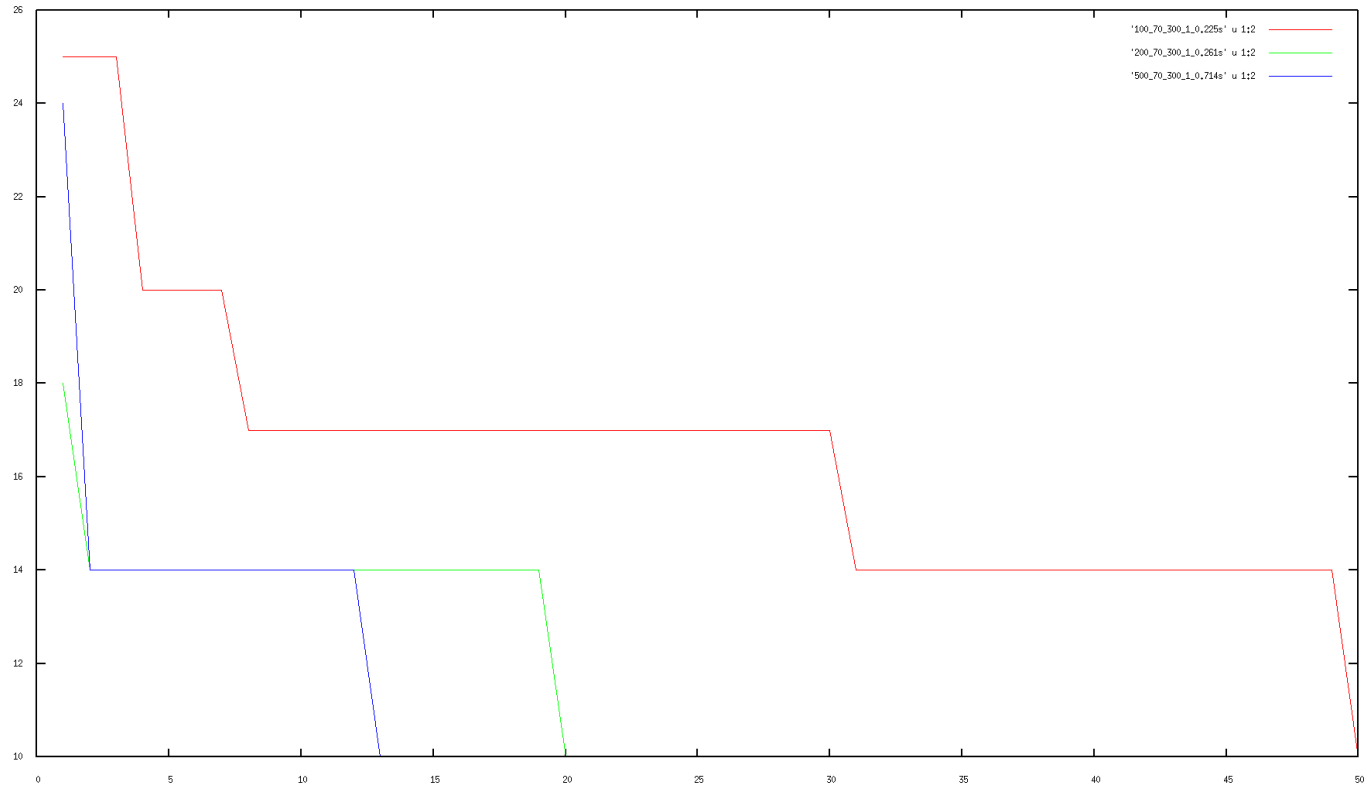


FIGURE 15 – Influence du nombre de génomes par génération (Matrice de taille 10)
axe des abscisses : numéro de la génération
axe des ordonnées : fitness du meilleur génome de cette génération

Sur ce graphe, on peut voir que toutes les simulations ont convergées vers un chemin de fitness de 10. Lorsque nous avons pris 100 génomes par génération, cela a été le plus rapide en temps mais cela a pris beaucoup de générations, ce qui signifie que si nous avions pris 40 générations au lieu de 50, nous n'aurions pas obtenu le chemin optimal pour 40 génomes par géneration. La taille de 200 semble optimale car elle converge rapidement en termes de générations et de temps. Cependant, on parle d'un temps inférieur à 1 seconde pour une matrice de taille 10 donc négligeable.

Pour la matrice de taille 25 :

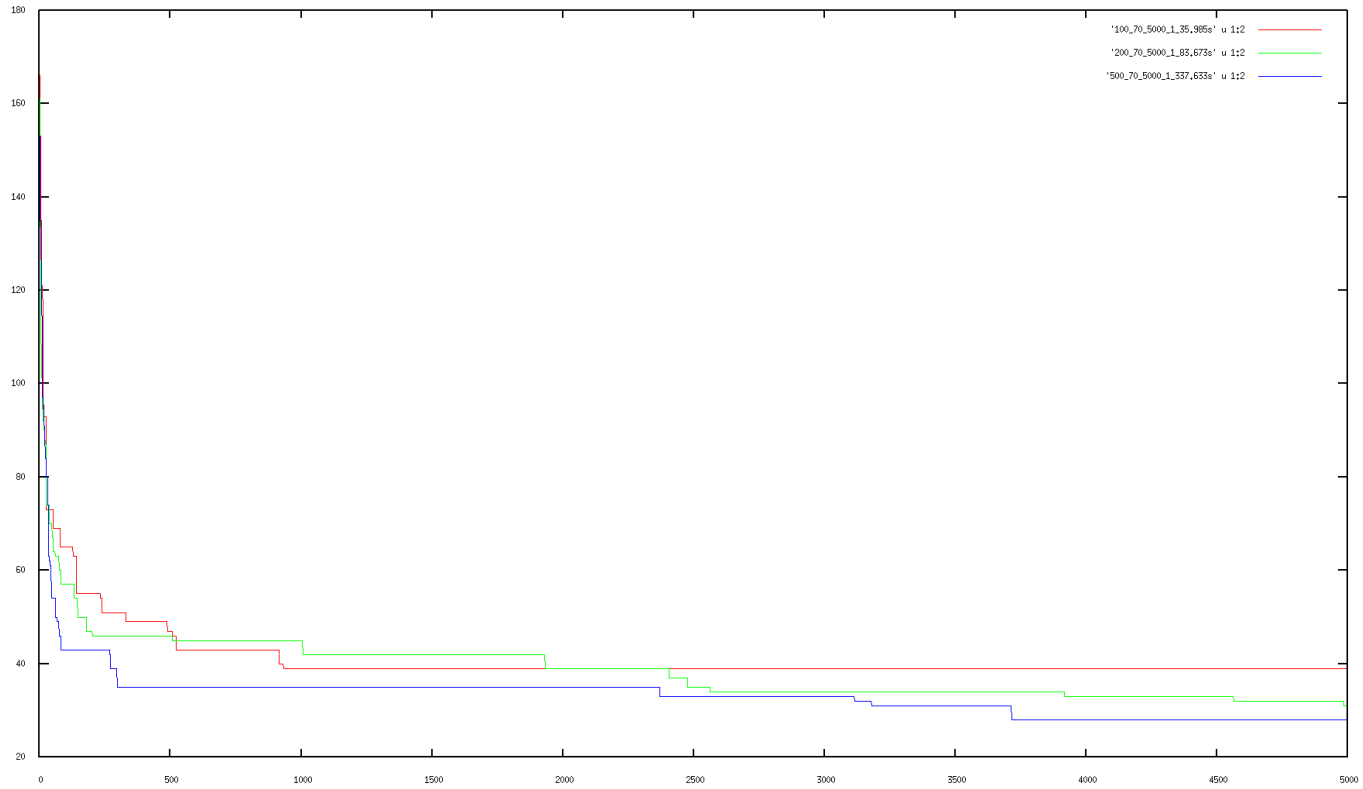


FIGURE 16 – Influence du nombre de génomes par génération (Matrice de taille 25)
axe des abscisses : numéro de la génération
axe des ordonnées : fitness du meilleur génome de cette génération

Voici les fitness du meilleur génome obtenue au bout 5000 générations et le temps d'execution :

- Pour une valeur de 500 génomes par génération : 28 en 338 secondes

- Pour une valeur de 200 génomes par génération : 31 en 84 secondes

- Pour une valeur de 100 génomes par génération : 39 en 36 secondes

Plus nous avons de génomes, meilleure est la solution obtenue. Cependant, si on compare 100 génomes et 500 génomes par génération, on a un gain de fitness de 28% pour 838% de temps en plus. Si on compare 200 génomes et 500 génomes par génération, la fitness est augmentée de 10% au prix de 307% d'augmentation du temps.

Nous nous rendons compte que le nombre de génomes par génération influe énormément sur le temps de compilation. Nous allons recommencer ces tests avec comme critère d'arrêt le temps écoulé et non plus le nombre de générations. Nous prendrons comme temps maximum 338 secondes et allons retester 500, 200 et 100 génomes par génération.
Comme nous avons du changer d'ordinateur, nous allons obtenir moins de générations pour un même temps car notre nouvel ordinateur est moins puissant.

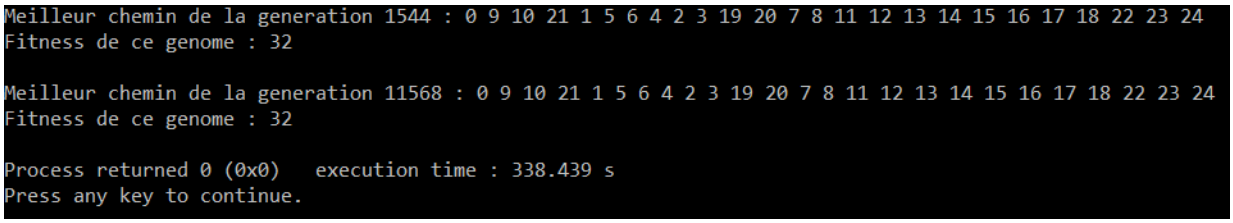


FIGURE 17 – 100 génomes par génération en 338 secondes

On se rend compte ici que nous n'avons pas eu d'amélioration entre la génération 1544 et la 11568, ce qui rend l'ajout de temps inutile. La fitness obtenue est meilleure mais cela est du "à la chance" et non aux paramètres, en effet elle a été obtenue à la génération 1544 alors que nous étions précedement allés jusqu'à la génération 5000 sans obtenir mieux qu'une fitness de 39.

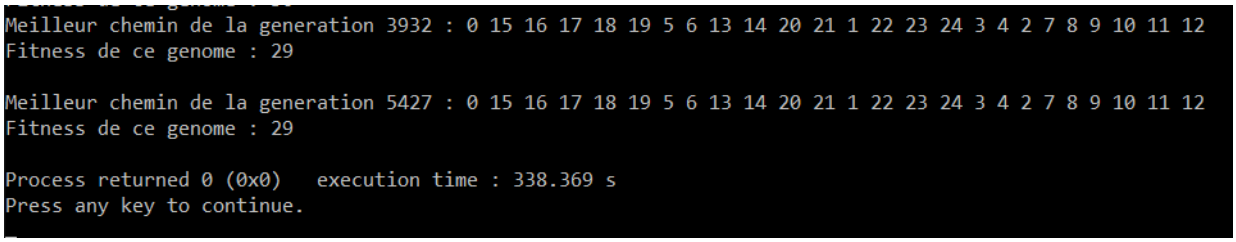


FIGURE 18 – 200 génomes par génération en 338 secondes

La fitness obtenue est meilleure, on a pu faire la moitié de générations qu'en taille 100.

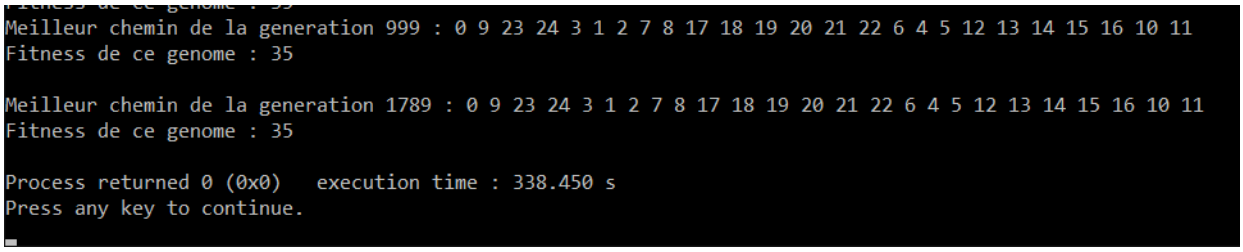


FIGURE 19 – 300 génomes par génération en 338 secondes

La fitness obtenu est moyenne. Cependant, on constate que on a fait 1789 générations ce qui est très faible. C'est du à notre ordinateur mais le rapport reste le même.

Conclusion

Le problème d'une taille importante est evidement le temps de compilation. Le problème d'une taille faible est que la fitness arrête de s'améliorer assez vite entre les génération 1000-1500, tandis que les tailles importantes ont une évolution plus lente qui peut continuer jusqu'à la génération 4000 et ainsi obtenir une meilleure fitness.

Pour la matrice de taille 50 :

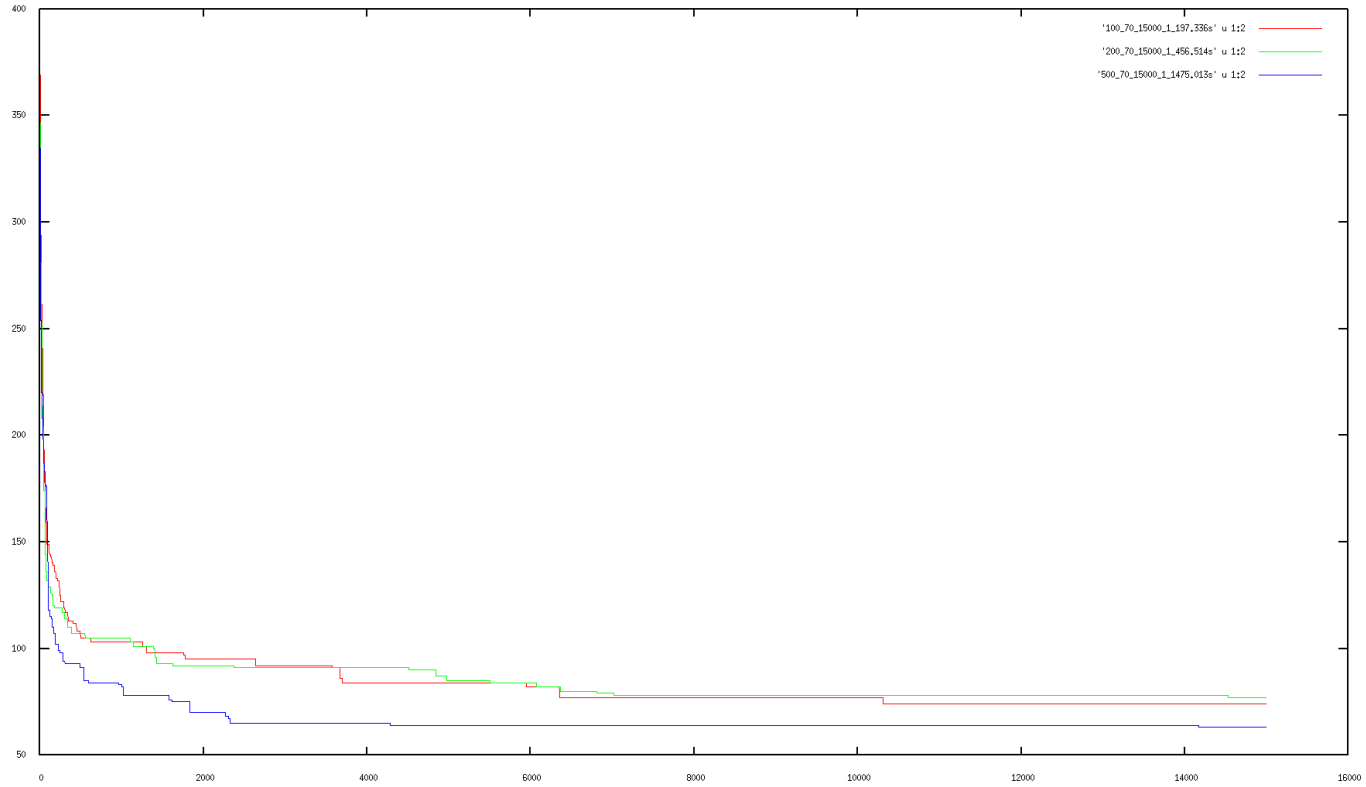


FIGURE 20 – Influence du nombre de génomes par génération (Matrice de taille 50)
axe des abscisses : numéro de la génération
axe des ordonnées : fitness du meilleur génome de cette génération

On constate encore ici que la taille de génération 500 obtient de meilleurs résultats que les autres, là encore au prix d'un temps nettement plus important : 1475 secondes pour une taille de 500 contre 455 pour 200 et 197 pour 100.
On constate aussi qu'on a eu une amélioration après la génération 14000 pour les tailles 200 et 500 ce qui sous-entendrait qu'on aurait eu encore d'autres améliorations si on avait décidé de faire encore plus de générations.
Néanmoins, nous ne l'avons pas fait car les temps de calcul étaient très longs. Il nous a semblé plus interressant de déterminer d'abord les autres paramètres optimaux comme le taux de mutation, le pourcentage de reproducteurs etc. Nous n'avons pas eu le temps d'y revenir.

5.3 L'influence du taux de mutation

Nous allons maintenant nous intéresser à l'influence du taux de mutation sur l'évolution de la fitness du meilleur génome pour une matrice de taille 25. Nous avons donc fixer les paramètres suivants :
- Nombre de génomes par génération = 500
- Pourcentage de reproducteurs = 60%
- Pourcentage d'enfants = 120%
- Méthode de sélection des reproducteurs : Eugénisme
- Pourcentage de conservation = 5%

Nous allons donc tester quatre valeurs différentes pour le taux de mutation : 25%, 50%, 75% et 100%.

Sur le graphe qui suit, la courbe bleue correspond à l'évolution de la fitness du meilleur génome en fonction de la génération pour la valeur de 25%, la rouge correspond à la valeur 50%, la jaune 75% et la verte a 100%.

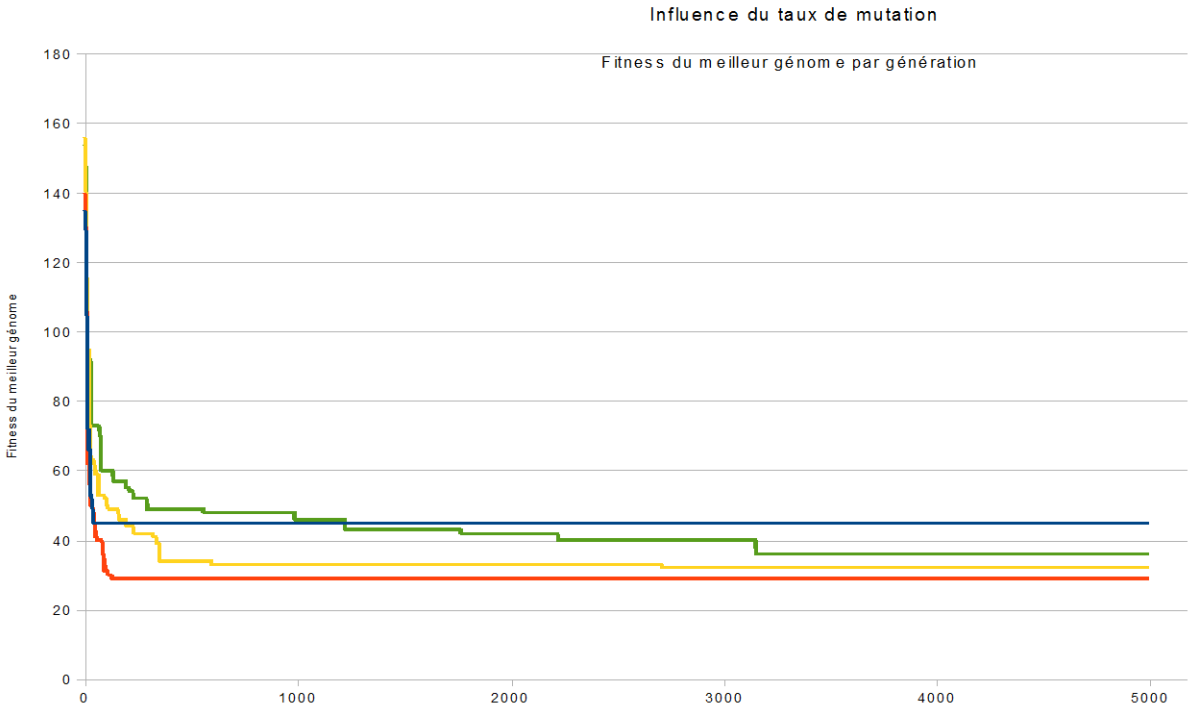


FIGURE 21 – Influence du taux de mutation (Matrice de taille 25)
axe des abscisses : numéro de la génération
axe des ordonnées : fitness du meilleur génome de cette génération

Plusieurs observations sont possibles. Commençons par la courbe bleue (taux de mutation égal à 25%), nous pouvons voir que la courbe décroit rapidement vers une valeur correcte mais stagne pendant très lontemps ensuite. Nous pouvons expliquer çela par le fait qu'avec ce faible taux de mutation, la convergence vers un minimum local est plus rapide, mais il est plus dur de sortir de ce minimum à cause du faible nombre de mutations.

Pour la courbe correspondant à la valeur de 50% (la rouge), nous pouvons tirer une conclusion similaire, à la différence que le minimum local trouvé est meilleur.

Pour les valeurs de 75 et 100%, qui sont de fortes valeurs, nous pouvons voir que les courbes convergent moins rapidement mais qu'il y a un plus grand nombre d'évolutions au cours de la simulation. Le grand nombre de mutations entraine une forte diversité. De plus, ces courbes continuent de décroître, elles arrivent mieux à se sortir d'un minimum local, car elles sont plus volatiles.

5.4 L'importance de la conservation

Il est très important d'avoir un pourcentage de conservation supérieur à 0%. Il ne faut pas non plus que ce dernier soit trop élevé pour ne pas nuire à l'évolution et converger trop vite vers un minimum local. Voici une courbe qui illustre cela. Il s'agit d'une courbe avec les mêmes paramètres que précédemment mais avec un pourcentage de conservation égal à 0%, le nombre de générations a également été réduit à 1000 et nous travaillons toujours sur une matrice de taille 25.

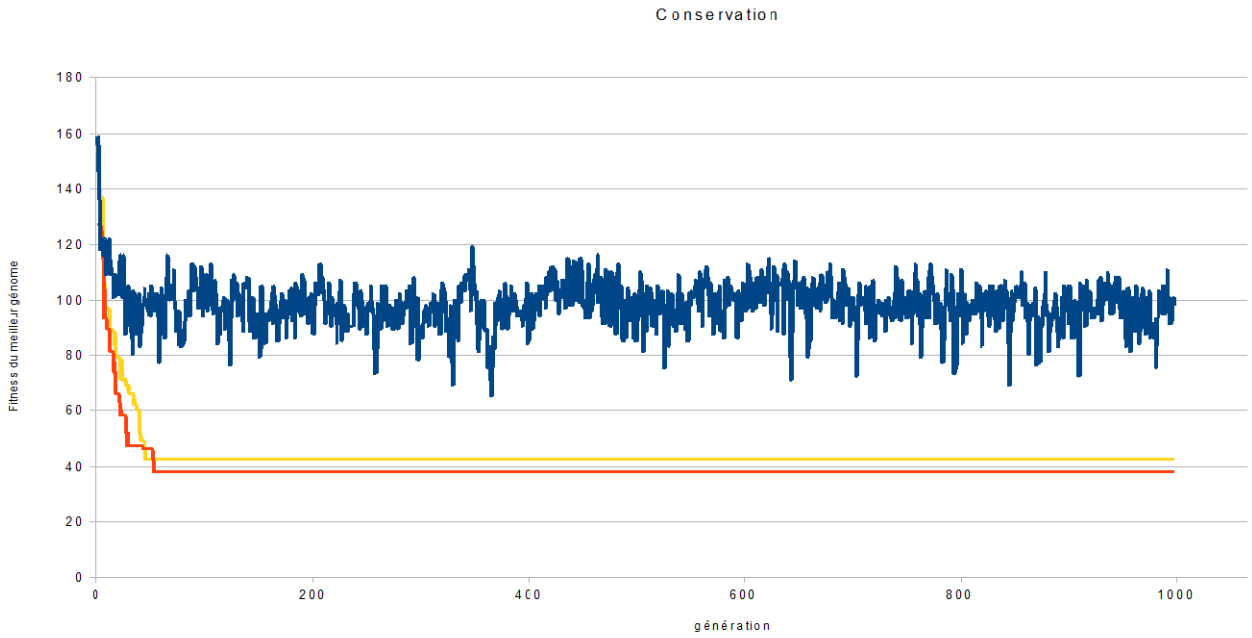


FIGURE 22 – L'importance de la conservation
axe des ordonnées : fitness du meilleur génome de cette génération

Nous pouvons donc voir qu'aucune convergence n'est envisageable. Nous n'avons donc pas retenu cette valeur pour le pourcentage de conservation. De plus pour des taux de conservation de 50% (courbe jaune) et 90% (courbe rouge) la convergence rapide vers minimum local (autour de'une fitness de 40) ainsi que l'absence d'amélioration sont bien illustrées.

5.5 Comparaison des méthodes de sélection de la génération suivante

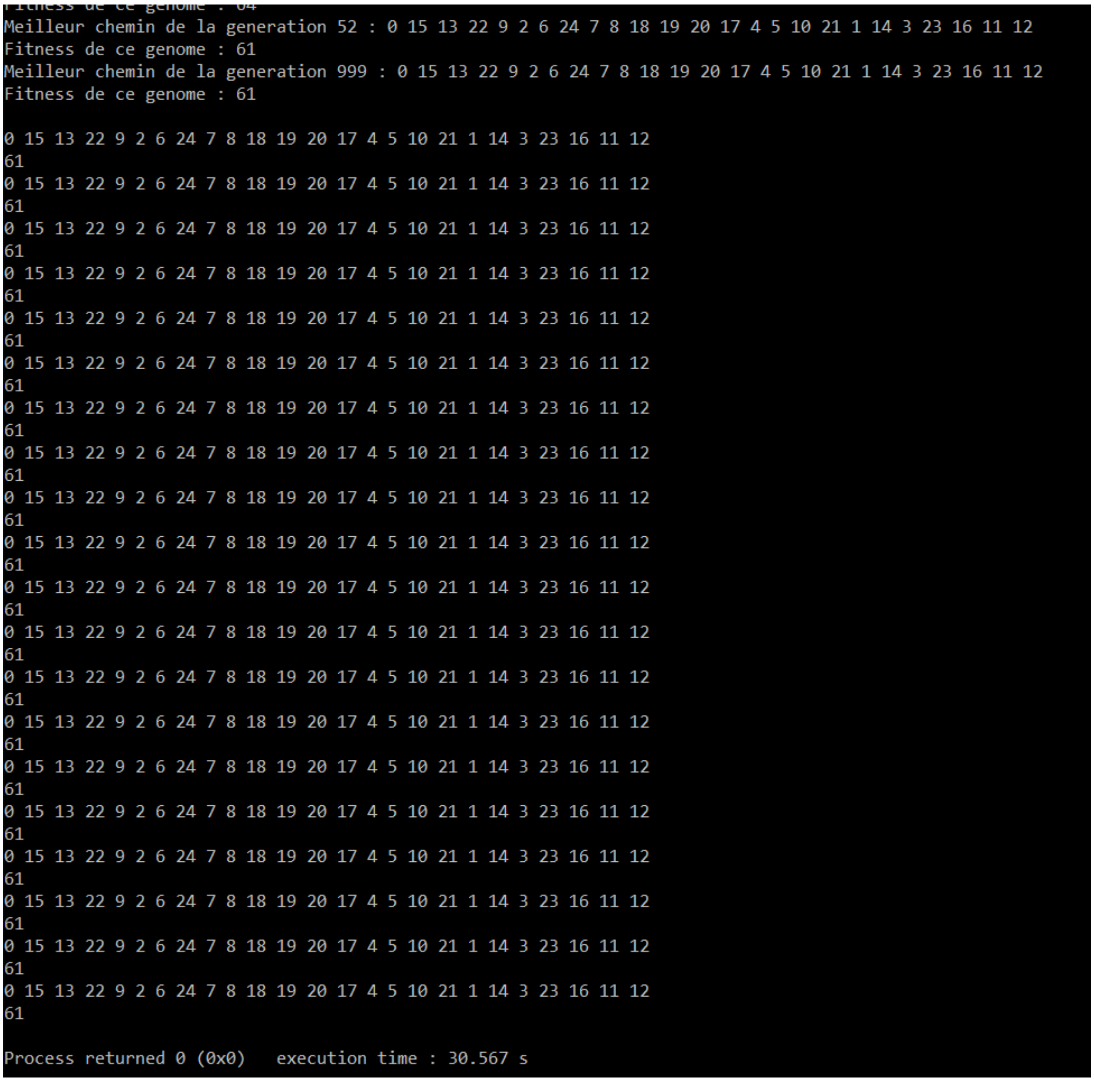
Nous rappellons que nous disposons de deux méthodes :

- Soit on prend un nombre fixe de génomes issus de la génération précédente (les meilleurs) et un nombre fixe d'enfants également issus de la génération précédente (les meilleurs enfants)

- Soit on prend les meilleurs génomes, qu'ils soient enfant ou non.

Nous allons montrer que la 2ème méthode ne fonctionne pas. Nous ne l'utiliserons plus par la suite.

Voici un test avec une taille de génération de 20, pour 25 villes, un taux de mutation de 70%, 60% de reproducteurs et 120% d'enfants.



```
Meilleur chemin de la generation 52 : 0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
Fitness de ce genome : 61
Meilleur chemin de la generation 999 : 0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
Fitness de ce genome : 61

0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61
0 15 13 22 9 2 6 24 7 8 18 19 20 17 4 5 10 21 1 14 3 23 16 11 12
61

Process returned 0 (0x0)   execution time : 30.567 s
```

Figure 23 – Exemple avec eugenisme total - 25 villes

Nous avons donc obtenu au bout de 1000 générations une fitness de 61 pour une taille de villes de 25 ce qui mauvais. Nous avons décider d'afficher à l'écran la population de la génération finale. Nous nous apercevons que tous les génomes sont identiques pour cette simulations ainsi que pour tous les autres tests que nous avons pu faire (avec plus de génomes par génération par exemple). Cela rend tous les cross-over inutiles et donc toute amélioration ne peut venir que de la mutation, ce qui n'est pas suffisant. C'est pourquoi nous avons mis de coté cette méthode.

6 Test globaux des paramètres

Nous avons vu que nous pouvons lister nos paramètres en entrée pour notre algorithme génetique :
- le nombre de génomes par génération
- le nombre de reproducteurs choisi
- le nombre de parents/d'enfants choisi pour former la génération suivante
- la méthode de sélection des reproducteurs
- le taux de mutation

Nous avons testé un par un les paramètres afin de démontrer les variations qu'ils impliquent. Nous aimerions trouver "la recette" parfaite, la meilleur combinaison de paramètres qui renvoie la meilleur fitness en moyenne. Nous avons déterminé théoriquement 3 méthodes :

- Une étude statistique

Puisque nous pouvons créer autant de résultats que nous le voulons. Nous pouvons utiliser des régréssions, des tests de corrélation, d'indépendance etc sur un ensemble de résultats pour essayer de déterminer les bons paramètres, pour déterminer quels paramètres fonctionnent le mieux, avec quelles méthodes etc. Nous n'avons pas retenu cette méthode car elle demande des connaissances en statistiques que nous n'avons pas.

- Un nouvel algorithme génétique

Pour chacun des paramètres en entrée, on peut définir une liste ou un intervalle de valeurs "interressantes" qui pourront être testées.

On peut ensuite redéfinir ce qu'est un individu, ce que sont les fonctions outils définies au début.

On peut former un individu comme une liste de paramètres. On définit un cross-over de la même manière : lecture des valeurs du premier parent jusqu'à un indice k puis lecture des valeurs du 2ème parent. Une mutation serait la modification d'un paramètre choisi aléatoirement vers une valeur aléatoire dans la liste des valeurs possibles pour ce paramètre. Enfin, la fonction d'évaluation est la fitness finale de notre algorithme génétique de résolution du problème du voyageur de commerce, prise avec ces paramètres d'entrées, le tout après n générations ou k secondes (au choix).

Nous n'avons pas eu le temps d'implémenter cette idée mais elle nous semble la plus prometteuse.

Elle nous permet aussi de démontrer que le principe général des algorithmes génétiques peut s'appliquer sur beaucoup de problèmes autres que celui du voyageur de commerce. Il faut cependant bien définir les fonctions outils.

- Un test empirique

C'est la méthode que nous avons employée. Nous avons défini des intervalles pour chaque paramètre :
- Le nombre de génomes par génération = [50, 75, 100, 150, 300, 500]
- Le pourcentage de génomes reproducteurs = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
- Le pourcentage de la population précédente conservée = [0, 5, 10, 15, 20]
- Le taux de mutation = [5, 15, 25, 35, 45, 55, 65, 75, 85, 95]
- Les 4 méthodes de sélection des reproducteurs : Eugenisme, Roulette, Rang, Tournoi, numérotée par 1, 2, 3 et 4. Nous avons retiré la sélection uniforme car elle a toujours donnée de mauvais résultats et nous voulions éviter les tests inutiles.

L'idée est de choisir des combinaison de paramètres, pris aléatoirement (par exemple : 50-30-20-95-1), de lancer n simulations de notre algorithme génétique pour ces paramètres. Le fait de lancer plusieurs simulations avec les mêmes paramètres en entrée nous permet de lisser nos résultats en évitant les valeurs aberrantes. Le critère d'arrêt choisi pour chaque simulation est un nombre de secondes. Cela nous permet de savoir à l'avance combien de temps va durer notre test. Nous allons donc écrire dans un fichier texte les meilleures combinaisons de paramètres obtenues au cours du test ainsi que la meilleur fitness obtenue avec ces paramètres.

Il y a en tout 12 000 possibilités de combinaisons (6 X 10 X 5 X 10 X 4) à multiplier par le nombre simulations pour chaque possibilité. On ne peut donc pas tout essayer. Toutefois, on peut faire plusieurs essais, essayer de reduire petit à petit nos intervalles de paramètres en entrée et avancer pas à pas vers la "recette idéale".
La "recette" dépend de la taille de la matrice en entrée. Nous avons donc fais notre étude pour une taille de 25, mais le raisonnement serait reproductible pour tout autre taille.

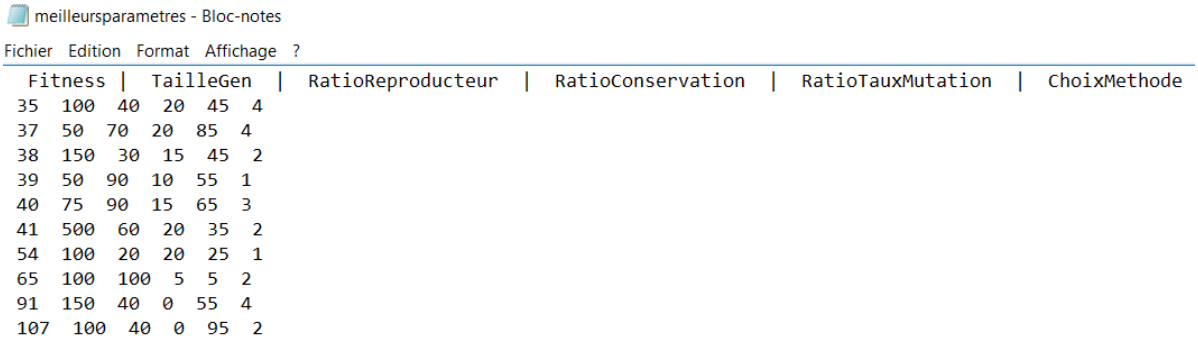


FIGURE 24 – Exemple de sortie pour 10 combinaisons, simulée 1 fois, avec 50 secondes chacune

La première colonne correspond à la meilleure fitness obtenue et ensuite nous avons les paramètres en entrée qui arrivent comme indiqué sur la première ligne.
La taille de l'echantillon étudié est trop petite pour conclure, elle fait office d'exemple.

Nous allons lancer une grosse simulation.

meilleursparametres(1).jpg - Bloc-notes

Fichier Edition Format Affichage ?

Fitness	TailleGen	RatioReproducteur	RatioConservation	RatioTauxMutation	ChoixMethode
31	500	60	10	75	2
32	300	60	15	75	1
32	100	10	5	55	2
32	100	60	10	65	1
33	500	60	5	75	2
33	500	100	10	55	4
33	150	50	10	75	2
33	100	60	10	75	4
34	300	60	5	65	4
34	300	50	10	85	1
35	100	70	5	65	1
35	50	70	15	95	4
35	300	40	5	45	2
35	500	80	10	55	1
35	75	50	15	85	1
35	500	90	10	85	1

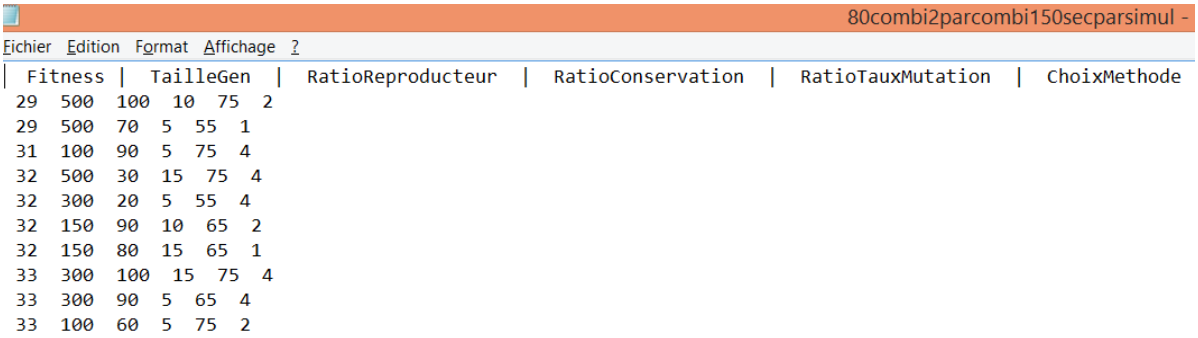
FIGURE 25 – Exemple de sortie pour 100 combinaisons, simulée 5 fois, durant 160 secondes chacune

```
Process returned 0 (0x0)   execution time : 80029.922 s
Press any key to continue.
```

FIGURE 26 – Temps de compilation pour la derniere simulation

Nous nous rendons compte que le temps de compilation est énorme. Par contre, les valeurs obtenues ici ont beaucoup de sens car nous avons les 16 meilleures sur 100 combinaisons. Nous voyons que le ratio de conservation n'est jamais 20% ou 0% mais plutôt 5,10,15. La taille de génération est entre 100 et 500, jamais 50 ou 75. Le taux de mutation est toujours compris entre 55 et 85. On ne peut pas conclure sur le ratio de reproducteurs. On peut aussi supprimer la méthode 3 qui n'apparait pas. Nous sommes volontairement expéditif dans nos choix d'intervalles car nous sommes pris par le temps et ces simulations sont très longues.
Nous pouvons donc recommencer sur ces nouveaux intervalles affinés de paramètres :
- Taille de la génération = [100, 150, 300, 500]
- Pourcentage Reproducteurs = [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
- Pourcentage de la population précedente conservé= [5, 10, 15]
- Taux de mutation= [55, 65, 75]
- Méthode = [1, 2, 4]

Nous avons donc maintenant 1080 combinaisons possibles au lieu de 12 000. Nous augmentons la probabilité de tomber sur la recette idéale.


80combi2parcombi150secparsimul -

Fichier Edition Format Affichage ?

Fitness	TailleGen	RatioReproducteur	RatioConservation	RatioTauxMutation	ChoixMethode
29	500	100	10	75	2
29	500	70	5	55	1
31	100	90	5	75	4
32	500	30	15	75	4
32	300	20	5	55	4
32	150	90	10	65	2
32	150	80	15	65	1
33	300	100	15	75	4
33	300	90	5	65	4
33	100	60	5	75	2

FIGURE 27 – 80 combinaisons, testées 2 fois et 150 secondes chacune

Cette simulation a pris environ 7 heures, les résultats sont moins concluant que la première. On remarque ques les seules simulations ayant atteint 29 de fitness possédaient 500 génomes par génération. On observe aussi que même si la méthode 4 n'a pas atteint 29, elle détient les places 3 et 4 avec égalité avec d'autres.

Nous ne pouvons pas conclure plus sur cette simulation. Il faudrait en lancer d'autres avec les mêmes critères pour éventuellement voir se dessiner une tendance.

Si nous avions eu plus de temps, nous aurions pu faire ceci avec la matrice de taille 50 et en donnant un temps plus important (300 secondes) par simulation. La taille plus importante de la matrice aurait mieux separé les simulations entre les bons paramètres et les moins bons.

7 Conclusion

Ce projet fut très instructif. Il nous a permis de comprendre le fonctionnement des algorithmes génétiques et à quel point ils peuvent être utilisé dans de très nombreux domaines. Nous pouvons même reprendre notre code à l'identique pour trouver la solution d'autre oroblème, il nous faudra alors modifer les fonctions outils et le "code génétique".

Nous avons utiliser une démarche scientifique pour tester chaque paramètre et ensuite affiner pas à pas vers les paramètres ideaux.

En termes de programmation, ce projet nous a permis de mieux appréhender les problèmes de mémoire, de performance et comment les gérer en C++.