Algorithme génétique

SOLUTION D'UN LABYRINTHE

DOUE – GABBAY – GHIGONIS - GONON | Algorithme & Programmation Objets |
07/01/2019

Table des matières

Présentation du sujet

L'objectif de ce projet est d'utiliser nos compétences acquises durant un semestre en algorithme et programmation afin de créer un programme simulant un cas réel. Pour ce dernier, nous avons choisi d'élaborer un algorithme génétique. La finalité de ce programme est de chercher une solution (approchée) à un problème difficile, voire impossible à résoudre par des méthodes classiques.

Le projet que nous avons choisi consiste à imiter le mécanisme d'évolution génétique d'une population. Ce mécanisme d'évolution a pour objet une **population** composée d'**individus**. Pour simuler l'évolution génétique d'une population, nous allons développer des modifications au sein d'une population pour en obtenir une nouvelle à l'aide de sélections, de croisements et de mutations. Pour simplifier le projet, nous allons considérer un **labyrinthe** que l'on devra créer et parcourir. Le chemin d'un individu dans ce labyrinthe constituera son génome (ensemble des gènes de l'individu).

Nos objectifs dans ce projet sont donc les suivants :

- Créer un labyrinthe
- Créer des individus (ayant chacun un génome qui correspond à un chemin du labyrinthe)
- Déterminer le meilleur individu (en calculant son score)
- Créer une population capable d'**évoluer** (sélection, croisement et mutation)

Si nous parvenons à résoudre ces différents objectifs, on aura créé un logiciel qui génère un labyrinthe et qui tente de résoudre le problème du chemin vers la sortie, en visualisant le comportement de différentes approches pour les opérations génétiques.

Nous utiliserons trois classes (VotreLabyrinthe, VotreIndividu, VotrePopulation) qu'on implémente des interfaces (Labyrinthe, Individu, Population) pour générer une interface graphique illustrant l'algorithme.

Analyse de la démarche-Synthèse

Définition des objets et diagramme de classe

L'algorithme génétique se base sur le fonctionnement d'un **labyrinthe** pour simuler les évolutions qu'une **population** d'**individus** peut subir.

Nos trois classes principales seront donc :

- VotreLabyrinthe
- VotreIndividu
- VotrePopulation

A présent, nous allons vous présenter les attributs de chacune de ces différentes classes :

VotreLabyrinthe :

- **int**[][] Labyrinthe
- **int**[][] chemin
- **int** nbLigne
- **int** nbColonne
- **int** ligneDep
- **int** ligneArr
- **Fenetre** fen

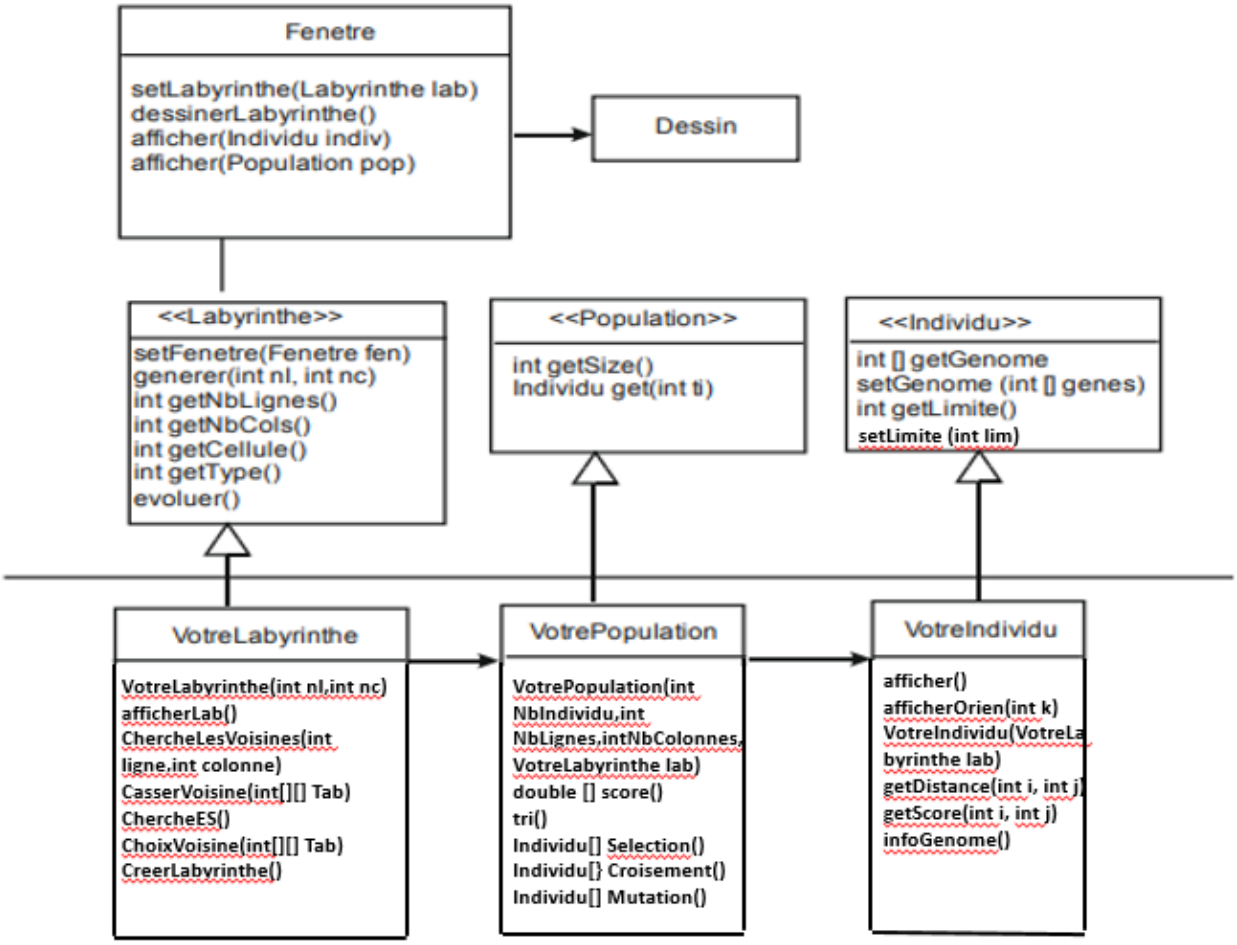
VotreIndividu :

- **int**[] genome
- **int** lim
- **VotreLabyrinthe** lab
- **int** nbl
- **int** nbc

VotrePopulation :

- **int** NbIndividu
- **VotreIndividu** Ind
- **VotreIndividu**[] Population

A présent, voici le diagramme de classe de notre algorithme :

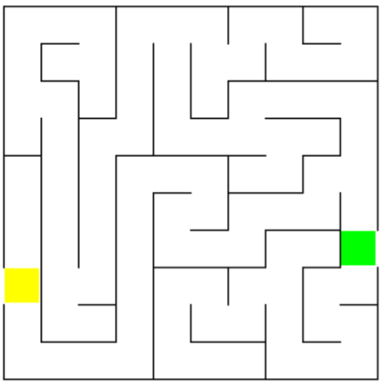


Création du labyrinthe

On crée la classe « VotreLabyrinthe » qu'on implémente à l'interface « Labyrinthe » afin de lier notre classe à cette dernière. On définit ensuite notre labyrinthe qui correspond à un tableau à double entrée composé de n lignes et m colonnes. Il est alors représenté de la façon suivante :

Chaque cellule est codée par un entier pouvant prendre comme valeur la somme d'une ou plusieurs des valeurs suivantes :

- 1 s'il existe un mur au Nord,
- 2 s'il existe un mur à l'Est,
- 4 s'il existe un mur au Sud,
- 8 s'il existe un mur à l'Ouest.



Pour créer le labyrinthe, on procède de la manière suivante :

On commence par créer un tableau à deux dimensions grâce au constructeur **VotreLabyrinthe(int nl, int nc)** dans lequel chacune de ses cases prend la valeur 15. Cela signifie que chacune des cases de notre labyrinthe de départ va posséder un mur au Nord, à l'Est, à l'Ouest et au Sud. Pour créer le labyrinthe, nous allons donc choisir une case aléatoirement dans le tableau a deux dimensions et on va la considérée comme étant visitée en lui affectant la valeur 1 dans le tableau à double dimension Chemin qui prend la valeur 1 si une case a été visitée en première et 0 sinon. La seconde case visitée prendra la valeur 2 dans le tableau Chemin et ainsi de suite.

Ensuite, nous allons déterminer les cases voisines à celle où l'on se situe en retournant leur coordonnées grâce à la méthode **ChercheLesVoisines(int ligne, int colonne).** Une fois que nous connaissons les cases voisines, nous allons procéder de la manière suivante :

- Si au moins une des cases voisines n'a pas encore été visitée, on choisit aléatoirement parmi elles laquelle on va visiter et on casse le mur entre la case précédente et la nouvelle à l'aide de la méthode **CasserVoisine(int[][] Tab)**

- Si toutes les cases voisines ont déjà été visitées, on choisira aléatoirement parmi toutes les cases voisines celle sur laquelle on se placera et vérifiera si elle possède des cases voisines non visitées. Si non, on réitère ce processus jusqu'à temps de trouver une case qui possède des cases voisines non visitées (le tout grâce à **ChoixVoisine(int[][] Tab)**).

Le labyrinthe sera créé lorsque le tableau Chemin aura une case qui prendra la valeur nbLignes*nbColonnes, ce qui signifie que chaque cellule du labyrinthe aura été visitée. Pour réaliser le labyrinthe, on va donc utiliser la méthode **generer(int nl, int nc)** qui utilise elle-même toutes les méthodes vues précédemment. Son principe est le suivant : on va définir un curseur qui verra sa valeur s'incrémenter d'un à chaque fois que l'on visite une nouvelle case. De ce fait, lorsque le curseur prendra la valeur nbLigne*nbColonne, cela signifiera que l'on aura parcouru toutes les cases du labyrinthe et donc cassé les murs comme nous l'avons décrit précédemment.

Il ne nous restera plus qu'à définir une cellule qui correspondra à l'entrée du labyrinthe et une autre correspondant à sa sortie. Pour ce faire, nous allons utiliser la méthode **ChercheES()** qui renvoie la ligne correspondant à la cellule d'entrée et une ligne correspondant à la cellule de sortie. Nous allons ensuite casser le mur situé à l'Ouest pour créer l'entrée du labyrinthe, et celui à l'Est pour la sortie.

Création d'individus et calcul de leur score

A présent, nous possédons le support de notre algorithme génétique puisqu'il permettra de simuler l'algorithme génétique. En effet, pour simuler l'évolution génétique d'une population, il faut créer les individus qui la constitueront. Or, chaque individu est caractérisé par un génome qui lui est propre. Et ce génome sera en réalité le chemin parcouru par un individu dans le labyrinthe.

Premièrement, pour créer un génome, nous créons un constructeur qui initialise une liste d'entiers aléatoirement choisis parmi {1,2,4,8} correspondant respectivement au Nord, à l'Est, au Sud, et à l'Ouest. La taille de la liste est définie comme le produit des dimensions du labyrinthe. Chaque entier désigne donc un gène.

Puis, comme le décrit le rapport, nous devons connaitre la portion de gènes valides (ici du chemin valide). Pour cela nous créons la méthode **InfoGenome()** qui retourne une liste d'entier correspondant respectivement à l'indice de la ligne de la dernière case valide, l'indice de colonne et le nombre de case que le chemin a fait avant d'atteindre cette limite. Ici, la limite correspond à la longueur du chemin valide effectuer par le génome.

Enfin nous calculons le score du génome en fonction de la case limite. La fonction **getScore()** retourne le score précédemment défini avec $\alpha = 0{,}5$. Cette fonction d'aide de **getDistance()** qui retourne la distance d'une case par rapport à la sortie en faisant une norme euclidienne avec comme coordonnées les indices des cases dans le labyrinthe.

Détail de l'algorithme InfoGenome() :

Premièrement, nous récupérons les coordonnées de la case de départ, puis nous supposons que le génome commence dès la case de départ. Ainsi la limite vaut au minimum la première case soit $lim_{min} = 1$. Puis, pour chaque gène du génome, on acquiert la valeur de la cellule (de la case) du labyrinthe où est sensé se situer le génome. Ensuite on étudie les murs présents de la cellule : au-dessus d'une certaine valeur, à l'aide des valeurs choisies pour coder le génome, on sait si un mur est présent ou non. Ensuite, le gène correspond à une valeur indiquant l'orientation à prendre. Enfin, si l'orientation est bonne, on passe à la prochaine cellule suivant l'orientation choisie, tout en augmentant la limite de $+1$. Sinon, l'algorithme s'arrête et on renvoie les indices de la dernière cellule limite et la limite.

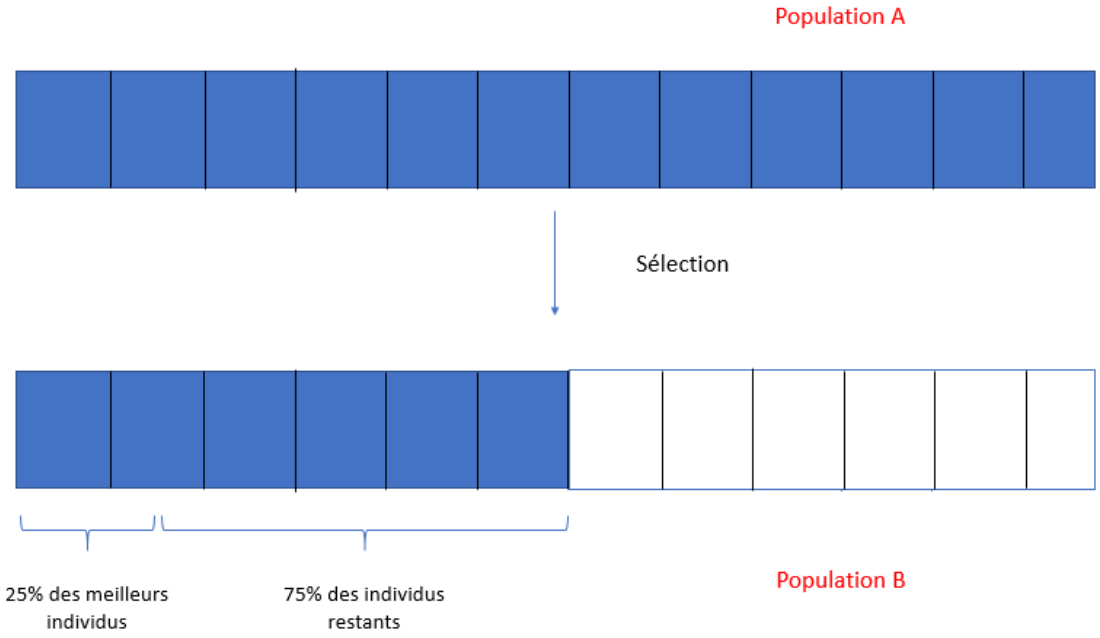
En conclusion, nous pouvons connaître le score de chacun des individus et ainsi savoir si certains d'entre eux ont un meilleur génome (c'est-à-dire qui sera plus apte à survivre lors d'évolutions génétiques).

Création d'une population soumise à l'évolution génétique

Maintenant que nous sommes capables de créer des individus et de déterminer leur score, nous pouvons former des groupes d'individus autrement appelées population. C'est ici que notre algorithme se concrétise puisqu'on va faire évoluer une population de manière génétique à l'aide de trois mécanismes :

- La sélection
- Le croisement
- La mutation

Pour simuler une évolution génétique, nous voulons créer une population B à partir d'une population A. La population B devra comporter la moitié des individus de la population A puisque les autres seront victime d'une sélection naturelle. Pour simuler cette sélection, nous allons créer une population B comprenant 25% des meilleurs individus de la population A et 75% d'autres individus de la population A. Pour obtenir les meilleurs individus de la population initiale, nous allons créer la méthode **score()** qui permet de retourner une liste comprenant le score de chaque individu à l'aide de la méthode **getScore**() de la classe VotreIndividu. Ensuite, nous allons trier cette liste à l'aide d'un algorithme de tri (nommé **tri()**) qui est efficace même s'il possède une complexité temporelle élevée ($O(n^2)$). Une fois la liste de score triée, nous pouvons créer la méthode **Selection()** qui va renvoyer une population B composée de 25% des meilleurs individus (les premiers dans la liste de la méthode **score()**). Ensuite, on choisit aléatoirement 75% d'individus parmi les individus restant de la population A. On obtient donc, à l'issue de la méthode de sélection, une population B composée de la moitié d'individus de la population A.



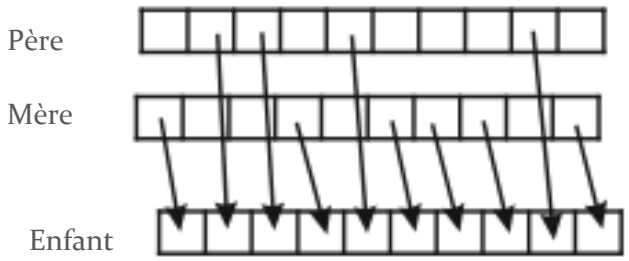
Une fois la sélection effectuée, il faut réaliser le mécanisme de croisements pour simuler la procréation des nouveaux individus de la population B. Pour ce faire, nous

allons faire **l'hypothèse simplificatrice** que chaque couple d'individu de la population B sera choisi aléatoirement et qu'ils n'auront qu'un seul enfant à la fois. De ce fait, nous allons générer autant d'individus (enfants) que nécessaire afin d'obtenir le même nombre d'individus que dans la population initiale (d'où l'utilisation d'une boucle for qui s'itère NombreIndividu/2 fois). Ainsi nous allons créer la méthode **Croisement()** qui va générer autant d'enfants que nécessaire pour remplir la population B.

Détail de l'algorithme de croisement :

Tout d'abord, pour créer un enfant, on va choisir aléatoirement dans la population B le père et la mère de l'enfant qui seront représentés par leurs indices dans la liste de la population. Ensuite, nous allons nous assurer que le père est bien différent de la mère (c'est-à-dire qu'ils n'aient pas le même indice) sans quoi on impose que la mère sera l'individu situé juste avant ou juste après le père dans la population. Par la suite, nous créons leur enfant grâce au constructeur d'individu par défaut. Nous allons donc maintenant devoir définir le génome de l'enfant à l'aide du génome du père et de la mère. Pour ce faire, nous allons faire **les hypothèses simplificatrices** suivantes :

- Le génome de l'enfant comportera autant de gènes que celui de ses parents (pas de cas de trisomie par exemple)
- On choisit aléatoirement pour chaque gène de l'enfant s'il provient du père ou de la mère.



Ainsi, on réalise une boucle for qui s'itère autant de fois qu'il y a de gènes dans le génome de l'enfant, et on complète donc chacun de ses gènes avec :

- Le gène du père correspondant si on tire un nombre au hasard dans [0 ;1[et qu'il appartient à [0 ;0,5[
- Le gène de la mère correspondant si on tire un nombre au hasard dans [0 ;1[et qu'il appartient à [0,5 ;1[

Une fois la méthode de croisement effectuée, on obtient en sortie une population B de la même taille que la population initiale.

Nous savons que le mécanisme de mutation qui consiste à modifier la valeur de certains gènes chez des individus existe dans la réalité. En revanche, la mutation intervient assez rarement sans quoi la population ne pourrait tendre vers une solution à un moment donné. Ainsi, nous allons faire **l'hypothèse** que la mutation modifie 1% des gènes des individus de la population B.

Détail de l'algorithme de mutation :

La méthode **Mutation()** consiste à parcourir chacun des gènes de tous les individus de la population B et de déterminer s'ils vont muter ou non. Pour cela, on utilise une première boucle for qui parcourt tous les individus de la population. Puis, on utilise une deuxième boucle for imbriquée qui parcourt chacun des gènes de chacun des individus. Pour savoir si un gène va muter ou non, on tire un nombre compris entre 0 et 1, et si ce nombre est supérieur à 1%, alors le gène ne mute pas, sinon il mute. Dans le cas où le gène mute, on lui affecte de manière une valeur correspondant à son orientation dans le labyrinthe.

Une fois tous ces mécanisme mis en œuvre, nous sommes en mesure de faire évoluer une population A vers une population B en simulant un mécanisme d'évolution génétique existant dans la réalité.

Mode d'emploi

```
public class Main {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        /*Scanner sc = new Scanner(System.in);
        System.out.print("Le nombre choisi est
: "+sc);*/
        /*
        //TEST VotreLabyrinthe
        Fenetre fen = new Fenetre();
        VotreLabyrinthe parcours = new
VotreLabyrinthe(10,10);
        fen.setLabyrinthe(parcours);
        //parcours.afficherLab();
        //parcours.CreerLabyrinte();
        //fen.setLabyrinthe(parcours);
        //parcours.afficherLab();
        //System.out.println();
        // TODO code application logic here */

        /*
```

On commence d'abord par créer un labyrinthe ici de taille 10*10grâce a la classe Votrelabyrinthe qu'on affiche à l'aide de l'interface graphique pour vérifier le bon fonctionnement du programme.

```
        int nl=5,nc=5;
        Fenetre fen = new Fenetre();
        VotreLabyrinthe lab= new
VotreLabyrinthe(nl,nc);
        fen.setLabyrinthe(lab);
        lab.CreerLabyrinte();
        VotreIndividu ind = new
VotreIndividu(lab);
        System.out.print(lab.getCellule(3,
3));
        //System.out.println();
        ind.afficher();
        int[] liste = ind.InfoGenome();
        System.out.println("Score :
"+ind.getScore(liste[0], liste[1]));
```

Pour la suite on passera à un labyrinthe de taille 5*5.

On vérifie le codage de la classe VotreIndividu en créant un individu issu dudit labyrinthe et en affichant son génome et son score en utilisant respectivement les méthodes afficher() et getscore().

```
        /*VotrePopulation pop=new
VotrePopulation(l,nl,nc,lab);
        pop.Afficher();
        System.out.print("\nAprÃ¨s selection
:");
        pop.Selection();
        pop.Afficher();
        System.out.print("\n");*/
```

On passe enfin à la creation d'une population costituée de 10 individus provenant du meme labyrinthe .D'abord on affiche chauqe individu de la population avec la methode Afficher()

Et on realise la selection des meilleurs individus de la population avec la methode Selection()

```
        for (int k = 0; k<10;k++){
        pop.Mutation();
        }
        pop.Afficher();
        pop.AfficherScore();
        for (int k = 0; k<50;k++){
        pop.Mutation();
        }
        pop.Afficher();
        pop.AfficherScore();
        for (int k = 0; k<100;k++){
        pop.Mutation();
        }
        pop.Afficher();
        pop.AfficherScore();
        for (int k = 0; k<500;k++){
        pop.Mutation();
        }
        pop.Afficher();
        pop.AfficherScore();

        //pop.tri();
        //pop.Afficher();
    }
}
```

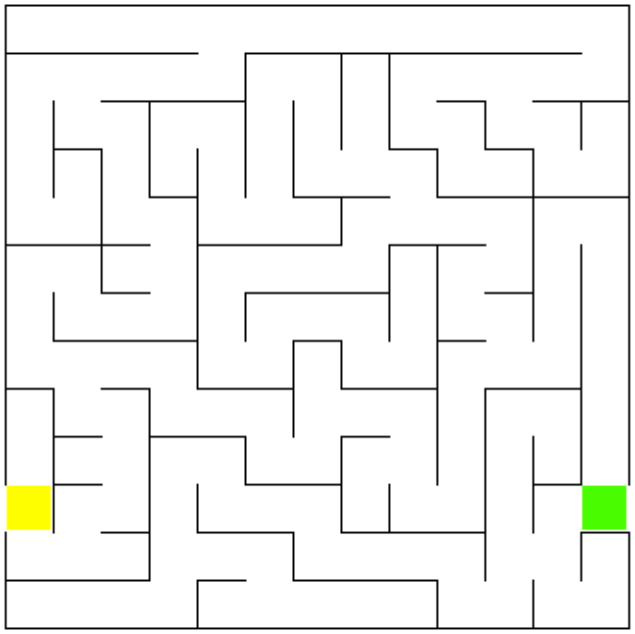
Pour finir ,on passe aux mutation sur plusieurs generations avec la methode Mutation() et on affiche l'etat de la population et le score des individus apres chaqu'une des mutations .

Tests de jeu

Création du labyrinthe et interface graphique

Tout d'abord, nous avons réussi à créer de multiples labyrinthes différents, en pouvant jouer sur leur taille.

Par exemple, voici ce qu'on obtient lorsque l'on créé un labyrinthe de taille 13x13 :



Les tableaux Labyrinthe et Chemin associés sont les suivants :

```
13 5 5 5 1 5 5 5 5 5 5 5 3
9 1 5 5 6 9 3 11 9 5 1 5 6
10 12 3 9 3 10 10 10 12 3 12 3 11
10 11 10 14 10 10 12 4 3 12 7 12 6
12 6 12 3 12 4 7 9 4 5 3 9 3
9 3 13 2 9 5 5 6 11 9 6 10 10
10 12 5 6 10 9 5 3 10 12 3 10 10
12 1 5 3 12 6 11 12 6 9 4 6 10
11 12 3 12 5 3 8 5 3 10 9 3 10
10 13 2 9 3 12 6 9 2 10 10 14 10
2 9 6 10 12 5 3 14 12 6 10 9 4
12 4 7 8 5 3 12 5 5 3 8 2 11
13 5 5 6 13 4 5 5 7 12 6 12 6

1 2 3 4 5 139 140 141 142 143 144 145 146
133 9 8 7 6 124 123 166 156 155 149 148 147
134 10 11 131 130 125 122 165 157 158 150 151 154
135 138 12 132 129 126 121 120 119 159 160 152 153
136 137 13 14 128 127 164 104 103 102 101 54 55
20 19 167 15 108 107 106 105 118 99 100 53 56
21 18 17 16 109 112 113 114 117 98 97 52 57
22 23 34 35 110 111 163 115 116 49 50 51 58
33 24 25 36 37 38 41 42 43 48 94 95 59
32 168 26 78 77 39 40 161 44 47 93 96 60
31 28 27 79 76 75 74 162 45 46 92 62 61
30 29 85 80 86 87 73 72 71 70 67 63 66
84 83 82 81 169 88 89 90 91 69 68 64 65
```

On remarque que le tableau Chemin comporte la case 169=13x13 ce qui signifie que toutes les cases du tableau ont été visitée et donc que le Labyrinthe a bien été créé.

Calcul du score d'individus

Nous avons aussi réussi à calculer le score des individus, ce qui est indispensable pour pouvoir ensuite savoir quels individus sont plus aptes que les autres à survivre aux sélections génétiques.

Par exemple, voici une simulation du score d'une population de 10 individus sur un labyrinthe de taille 5x5 :

```
0.7562277660168379
0.762842712474619
0.7228427124746191
0.762842712474619
0.762842712474619
0.762842712474619
0.762842712474619
0.762842712474619
0.3949509756796392
0.751547594742265
```

Evolution d'une population d'individus

Le but final de notre algorithme était de simuler l'évolution génétique d'une population d'individus. Et c'est ce que l'on va réaliser à travers l'exemple suivant :

Le test sera effectué sur une population de 10 individus et un labyrinthe de taille 5x5 :

A gauche sera représenté le chemin parcouru par chacun des individus et à droite leur score :

Avant la première génération :

```
Sud Nord Ouest Est Sud Est Ouest Nord Sud Sud Est Nor      0.7228427124746191
Ouest Ouest Ouest Nord Nord Sud Nord Sud Sud Est Oues      0.762842712474619
Nord Nord Sud Ouest Sud Est Sud Est Est Nord Sud Oues      0.762842712474619
Ouest Nord Est Est Ouest Sud Ouest Ouest Nord Ouest (      0.762842712474619
Nord Sud Nord Est Nord Nord Sud Nord Sud Nord Est Oue      0.762842712474619
Sud Nord Sud Est Sud Sud Est Sud Nord Est Sud Est Est      0.5949509756796392
Ouest Sud Ouest Nord Ouest Nord Est Sud Est Ouest Suc      0.762842712474619
Est Est Ouest Ouest Ouest Est Sud Nord Sud Nord Sud S      0.682842712474619
Sud Nord Ouest Ouest Ouest Ouest Nord Nord Ouest Oues      0.7228427124746191
Ouest Est Sud Nord Est Ouest Sud Est Sud Ouest Est Su      0.762842712474619
```

Après 20 générations :

```
Sud Nord Est Est Sud Nord Sud Ouest Nord Nord Sud N        0.5121320343559642
Sud Nord Est Est Sud Sud Sud Ouest Nord Ouest Sud N        0.5335533905932737
Ouest Nord Est Est Sud Nord Sud Ouest Nord Ouest Es        0.762842712474619
Ouest Nord Est Est Sud Nord Sud Ouest Nord Ouest Ou        0.762842712474619
Sud Nord Est Est Sud Sud Sud Ouest Nord Ouest Sud N        0.5335533905932737
Sud Nord Est Est Sud Nord Sud Ouest Nord Ouest Sud         0.17071067811865476
Sud Nord Est Est Sud Nord Sud Ouest Nord Ouest Sud         0.22142135623730952
Ouest Nord Est Est Sud Nord Sud Ouest Est Ouest Oue        0.762842712474619
Sud Nord Est Est Sud Nord Sud Ouest Nord Ouest Sud         0.17071067811865476
Ouest Nord Est Est Sud Nord Sud Ouest Nord Ouest Ou        0.762842712474619
```

Après 100 générations :

```
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.15811388300841897
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.22142135623730952
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
Sud Nord Est Est Sud Est Sud Ouest Nord Ouest Sud Nord Nc  0.13071067811865475
```

On se rend bien compte que, au fur et à mesure que le fait évoluer notre population de départ, tous les individus tendent à être les mêmes car leur génome devient quasi-identique au bout de 100 générations. Ainsi, on retrouve le fait que notre algorithme génétique permet à une population de tendre vers un extremum local c'est-à-dire une population devient de plus en plus résistante à l'évolution génétique (et donc que ses individus parviennent à trouver le chemin vers la sortie du labyrinthe).