

HE-Arc INF3-dlmb

# Rapport TP IA

Algorithmes génétiques

Monnet Joris  
29/11/2020

## Table des matières

Introduction.....	2
Choix lors de l'implémentation .....	3
Génération du labyrinthe : .....	3
Affichage des labyrinthes : .....	3
Implémentation de l'algorithme : .....	3
Encodage : .....	5
Fitness : .....	5
Sélection : .....	5
Crossover : .....	6
Mutation : .....	6
Arrêt de l'algorithme : .....	6
Population : .....	6
Résultats : .....	7
Grille 10x10 : .....	7
Grille 15x15 : .....	8
Grille 20x20 : .....	9
Grille 30x30 : .....	10
Grille 40x40 : .....	11
Conclusion : .....	12
Sources : .....	12

## Introduction

Ce TP propose de mettre en place un algorithme génétiques pour trouver un chemin. Le plateau est en réalité une grille d'une taille  $M \times N$  où chaque case à deux valeurs possibles, 0 ou 1, libre ou mur. Le but de notre algorithme est de trouver un chemin en ne passant que par des cases libres du point (0,0) au point (M-1, N-1). Les tailles de grille à tester sont déjà définies avec un temps maximum d'exécution pour chaque taille de grille :

Taille	Temps max
10x10	10s
15x15	15s
20x20	30s
30x30	60s
40x40	90s

Dans le cadre de ce TP je ne testerai donc que des grilles où  $M = N$ . De plus, le Framework DEAP sera utilisé.

Tout d'abord je vais vous présenter les choix que j'ai réalisés lors de l'implémentation de l'algorithme, notamment au niveau de ses paramètres. Je montrerai ensuite les résultats obtenus pour différentes grilles.

## Choix lors de l'implémentation

### Génération du labyrinthe :

Afin de générer un labyrinthe, j'ai utilisé une fonction pour créer le labyrinthe aléatoirement suivant une taille MxN avec 10% de 1 et 90% de 0. Je rappelle que les cases 0 sont les cases libres et les cases avec mur sont les cases 1. Ce labyrinthe est interprété par le programme comme étant une matrice.

### Affichage des labyrinthes :

J'ai gardé l'implémentation fourni avec des cases blanches lorsqu'elles sont libres, des cases grises lorsqu'il s'agit d'un mur et des cases bleues pour montrer le chemin.

### Implémentation de l'algorithme :

D'abord, on génère la grille aléatoire, Ensuite on crée une population, on l'évalue, on sélectionne via les tournois (10 ici) des membres de la population, on accouple ces individus pour donner naissance à des enfants, on en mute certains et cela en boucle jusqu'à atteindre notre objectif. D'un point de vue code, d'abord on lance les fonctions utiles de DEAP :

```
#DEAP functions
#creator
creator.create("FitnessMin", base.Fitness, weights=(-10000.0, -10.0))
creator.create("Individual", list, fitness=creator.FitnessMin)

#toolbox from Base
toolbox = base.Toolbox()
toolbox.register("fitness", fitness)
toolbox.register("mate", tools.cxMessyOnePoint)
toolbox.register("mutate", tools.mutUniformInt, low=0, up=MAX_ENCODING, indpb=0.1)
toolbox.register("select", tools.selTournament)
toolbox.register("init_gene", random.randint, 0, MAX_ENCODING)
toolbox.register("init_individual", tools.initRepeat, creator.Individual, toolbox.init_gene, CHROMOSOME_LENGTH)
toolbox.register("init_population", tools.initRepeat, list, toolbox.init_individual)
toolbox.register("evaluate", evaluatePopulation)
populationSize = 100
population = toolbox.init_population(n=populationSize)
toolbox.evaluate(population, end_cell)
```

On voit qu'ici la population est créée, évaluée une première fois.

Ensuite on met en place la boucle d'exécution des étapes listées ci-dessus :

```
while endTime < max_time_s and count < 5:
```

On sélectionne via le tournoi :

```
#tournament
children = toolbox.select(population, len(population), tournoisize)
children = list(map(toolbox.clone, children))
```

On met en place l'accouplement(crossover) avec un paramètre choisi :

```
#crossover
for child1, child2 in zip(children[::2], children[1::2]):
    if random.random() < CXPB:
        toolbox.mate(child1, child2)
```

Ensuite on mute certain Individus avec un paramètre choisi :

```
#mutation
for mutant in children:
    if random.random() < MUTPB:
        toolbox.mutate(mutant)
```

Enfin on évalue :

```
#evaluate children
population = children
toolbox.evaluate(population, end_cell)
```

Une fois un chemin trouvé, on le fait muter en essayant de trouver un meilleur chemin :

```
toolbox.register("mate", tools.cxOnePoint)
```

Enfin on récupère le winner, c'est-à-dire le seul chemin trouvé ou le meilleur des chemins trouvés ou l'essai qui s'est le plus rapproché du point d'arrivée :

```
winners = list(filter(lambda pop: pop.fitness.values[0] == 0, population))

if winners and len(winners)!=0:
    #result = the winner if he is alone or the shortest
    finalResult = winners[0] if len(winners) == 1 else list(reduce(lambda ind1, ind2: ind1 if len(getChromosomePath(ind1)) < len(getChromosomePath(ind2)) else ind2, winners))
else:
    #result = the best path found if doesn't find the path
    finalResult = population[np.argmin(np.array([ind.fitness.values[0] for ind in population]))]
    print("No winner found, show the best try")
```

Enfin, on finit l'algorithme par un affichage simple du temps de process, du nombre d'itérations, des paramètres et du résultat sur la grille.

## Encodage :

Les gènes sont encodés sur deux bits puisqu'il n'y a que quatre déplacements : gauche, droite, haut, bas. Dans mon cas j'utilise un namedtuple pour pouvoir directement associer un mouvement à une action sur une position. Je remercie M. Maxime Welcklen pour cette idée, très utile et efficace.

```
Move = namedtuple("Move", ["apply", "str"])
MOVES = {
    0: Move(lambda position: (position[0]-1, position[1]), "left"),
    1: Move(lambda position: (position[0]+1, position[1]), "right"),
    2: Move(lambda position: (position[0], position[1]-1), "top"),
    3: Move(lambda position: (position[0], position[1]+1), "bottom")
}
```

## Fitness :

J'utilise dans ma fonction de fitness une distance de Manhattan et je prends en compte la distance qu'il reste à parcourir en créant un coefficient. Le but du programme est donc de tendre vers 0, c'est-à-dire quand la position actuelle sera au bout de son voyage, au contraire elle vaut 1 lorsque la position actuelle est la position de start :

```
((float(manhattanDistance(target, currentPos)) / manhattanDistance(start_cell, currentPos))),
```

La fonction fitness consiste à appliquer le mouvement à une case et directement corriger si la valeur pour bouger aléatoire fait sortir de la grille ou lorsque le chemin passe dans un mur :

```
def fitness(individual, target):
    """ Fitness function of the chromosome :"""
    currentPos = start_cell #currentPos = the position we treat
    for i in range(len(individual)):
        gene = individual[i]
        nextPos = MOVES[gene].apply(currentPos) #nextPos = the next position after the move from the currentPos
        while nextPos[0] not in range(N) or nextPos[1] not in range(N) or grid[nextPos[0]][nextPos[1]] != 0: #value is not 0: #value is not 0
            individual[i] = random.randint(0, MAX_ENCODING) #create a random move
            nextPos = MOVES[individual[i]].apply(currentPos)
        currentPos = nextPos
        if currentPos == target: return (0, i+1) #return the number of iterations(+1 because currentPos = newPos)

    if start_cell == currentPos: return (50000,5000) #Not good because start cell = 0,0 so in the next line it will be 0/0
    return ((float(manhattanDistance(target, currentPos)) / manhattanDistance(start_cell, currentPos)), i+1)
```

Deux return avant la distance de Manhattan : le premier est simplement si on vient d'arriver à la dernière étape du chemin, c'est-à-dire au point d'arrivée. Le deuxième est la mise en place d'une valeur énorme (5000,5000) dans le return dans le cas où le point actuel serait le point de départ, en effet cela induirait une division par 0 dans le coefficient retourné avec la distance de Manhattan un ligne en dessous (si il n'y avait pas ce return).

## Sélection :

J'utilise un tournoi à 10. Ce paramètre est modifiable via la variable tournoisSize :

```
tournoisSize = 10
```

### Crossover :

Ce paramètre a été défini via des essais successifs, il est de 0.75 dans mon cas. Il est modifiable dans la variable CXPB :

```
CXPB = 0.75
```

### Mutation :

Ce paramètre a été trouvé via des essais successifs conjointement avec CXPB. Il est de 0.65 et est modifiable dans la variable MUTPB :

```
MUTPB = 0.65
```

### Arrêt de l'algorithme :

Le choix a été fait de s'arrêter lors de la fin du temps maximum alloué ou lorsqu'un chemin a été confirmé comme étant le plus court 5 fois.

```
while endTime < max_time_s and count < 5:
```

La variable count permet de voir combien de fois le plus court chemin a été confirmé :

```
fastestPath = np.min([ind.fitness.values[1] for ind in population if ind.fitness.values[0] == 0])  
if fastestPath == lastPath: count += 1  
lastPath = fastestPath
```

### Population :

La population est créée suivant une taille définie (100 dans mon cas) elle peut être modifiée via la variable populationSize :

```
populationSize = 100
```

Avec tous ces paramètres, je vais donc à présent vous présenter les résultats observés :

## Résultats :

Grille 10x10 :

▶ ▶ M

```
# Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 10)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3), (2,3), (3,3), (4,3),
#            (5,3), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8), (7,8),
#            (8,8), (9,8), (9,9)]
```

Found in 15 iterations

Found in 0.16755986213684082s

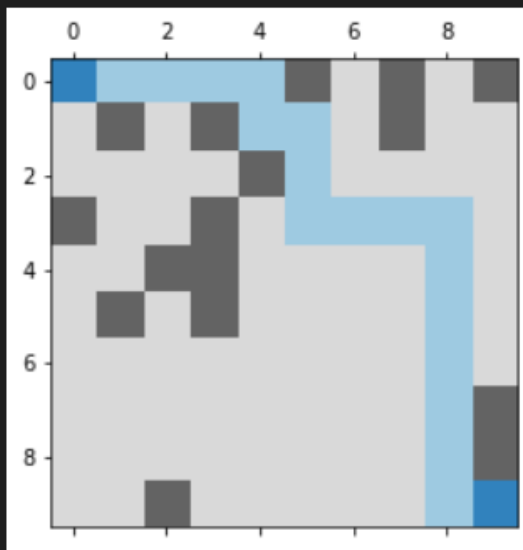
MUTPB=0.65 and CXPB=0.75

Size of the population : 100, with 10 selectionned per iteration

Length : 19

▶ ▶ M

```
display_labyrinth(grid, start, end, solution)
```





Grille 15x15 :

▶ ▶≡ Ml

```
# Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 15)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3), (2,3), (3,3), (4,3),
#            (5,3), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8), (7,8),
#            (8,8), (9,8), (9,9)]
```

Found in 13 iterations

Found in 0.15758085250854492s

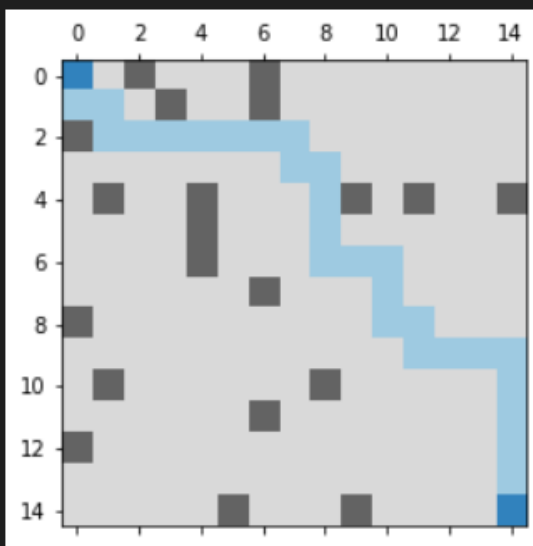
MUTPB=0.65 and CXPB=0.75

Size of the population : 100, with 10 selectionned per iteration

Length : 29

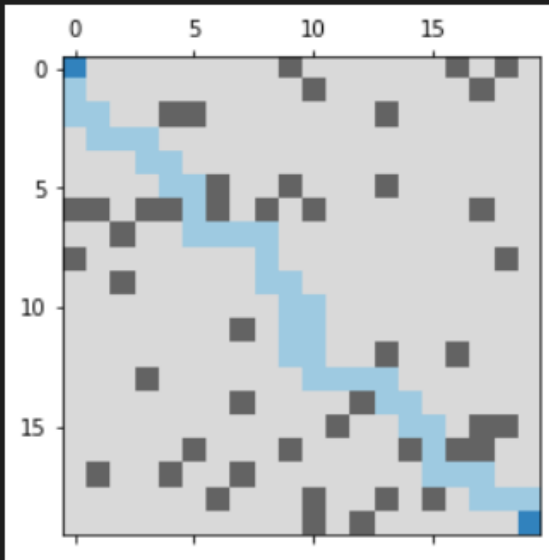
▶ ▶≡ Ml

```
display_labyrinth(grid, start, end, solution)
```



Grille 20x20 :

```
▶ ▶≡ M↓  
# Run the genetic algorithm  
solution = solve_labyrinth(grid, start, end, 15)  
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3), (2,3), (3,3), (4,3),  
#           (5,3), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8), (7,8),  
#           (8,8), (9,8), (9,9)]  
  
Found in 11 iterations  
Found in 0.1426222324371338s  
MUTPB=0.65 and CXPB=0.75  
Size of the population : 100, with 10 selectionned per iteration  
Length : 43  
  
▶ ▶≡ M↓  
display_labyrinth(grid, start, end, solution)  
  
▶ ▶≡ M↓
```



Grille 30x30 :

▶ ▶ M4

```
# Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 15)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3), (2,3), (3,3), (4,3),
#            (5,3), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8), (7,8),
#            (8,8), (9,8), (9,9)]
```

Found in 17 iterations

Found in 0.577455997467041s

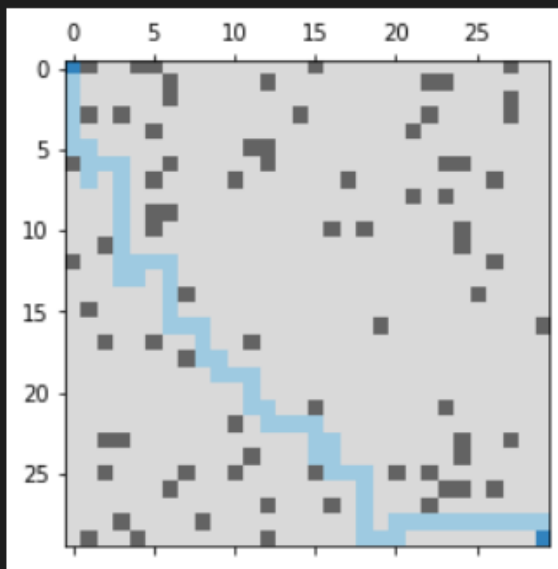
MUTPB=0.65 and CXPB=0.75

Size of the population : 100, with 10 selectionned per iteration

Length : 69

▶ ▶ M4

```
display_labyrinth(grid, start, end, solution)
```

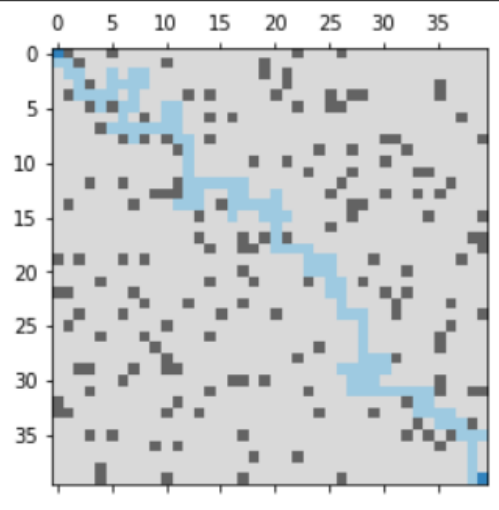


Grille 40x40 :

```
[44] ▶ ⌵ Ml
# Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 15)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3), (2,3), (3,3), (4,3),
#            (5,3), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8), (7,8),
#            (8,8), (9,8), (9,9)]

Found in 18 iterations
Found in 0.985152006149292s
MUTPB=0.65 and CXPB=0.75
Size of the population : 100, with 10 selectionned per iteration
Length : 189
```

```
[45] ▶ ⌵ Ml
display_labyrinth(grid, start, end, solution)
```



## Conclusion :

On peut voir au vu de ces résultats que les tailles 30x30 et 40x40 ne sont pas sur les chemins les plus courts à chaque fois notamment les deux exemples ci-dessus. Il est donc utile de trouver de meilleurs paramètres pour du pathfinding. Cela étant dit le but de ce TP était surtout de trouver des solutions en dessous d'un temps donné. On voit ici qu'on reste en dessous des 1 s pour toutes les tailles de grilles testées soit largement en dessous des limites accordées. Les résultats sont donc satisfaisants avec une marge de progression.

Ce TP aura permis de prendre en main les outils du Framework DEAP et de mettre en place un algorithme génétique. Dans un cadre plus grand, cela m'a donné des idées pour mon P3 qui étant une application de randonnée demande la mise en place d'algorithme de path finding.

## Sources :

Documentation officielle DEAP, aide de M. Welcklen cité plus haut,

[https://en.wikipedia.org/wiki/Genetic\\_algorithm](https://en.wikipedia.org/wiki/Genetic_algorithm)

<https://deap.readthedocs.io/en/master/examples/>

[https://www.researchgate.net/publication/235707001\\_DEAP\\_Evolutionary\\_algorithms\\_made\\_easy](https://www.researchgate.net/publication/235707001_DEAP_Evolutionary_algorithms_made_easy)