haute école
neuchâtel berne jura
arc ingénierie
www.he-arc.ch

Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts
Western Switzerland

Cours Intelligence Artificielle

TP Algorithme Génétique

Résolution de Labyrinthe

Adrien Paysant
24/11/2020

Table des matières

Introduction

Le but d'un algorithme génétique est d'approcher la solution d'un problème impossible à résoudre de manière exacte dans un délai raisonnable.

Nous pouvons trivialement résumer la finalité des algorithmes génétiques comme étant la solution à la recherche d'une aiguille dans une botte de foin. En effet un algorithme génétique ne fonctionnera pas à 100% mais lorsqu'il fonctionne, son temps d'exécution menant à une solution est très largement inférieur à une méthode sûre.

Afin d'appliquer les connaissances théoriques acquise lors du cours d'intelligence artificielle, nous allons mettre en œuvre la résolution de chemin au sein de labyrinthe via l'emploi du Framework DEAP en python. Le labyrinthe verra son départ en son coin supérieur gauche et son arrivée sur le coin inférieur droit.

Dans une première partie nous expliciterons les choix de paramètres pour l'algorithme (encodage, fitness, génération de labyrinthe, etc. …).

La seconde partie présentera quelques résultats d'expériences.

Choix des paramètres

Encodage :

Les gènes sont sur 2 bits (HAUT/DROITE/BAS/GAUCHE) qui permettent de couvrir tous les déplacements.

Les chromosomes sont de longueur fixe en fonction de la grille (si la grille est connue on utilise des valeurs arbitraires qui fonctionne bien, sinon on prend le nombre de case vide du labyrinthe).

Fitness :

La distance de Manhattan entre le dernier point du chemin et l'arrivée (coin Sud Est) ainsi que la longueur du chemin sont pris en compte.

Sélection :

Au travers d'un tournois à 10.

Crossover :

En un point avec une part variable de la population en fonction de la taille de la grille. Ce paramètre a été déterminé par des essais successifs.

Mutation :

Avec l'outil mutFlipBit avec une probabilité à 10% pour une part variable de la population en fonction de la taille de la grille. Ce paramètre a été déterminé par des essais successifs.

Critères Stop :

On s'arrête quand le temps est écoulé ou que l'on a une solution.

Création population :

La population est créée comme une suite de 0 et 1 aléatoire avec une taille préalablement fixée.

Déroulement de l'AG :

1-Créer la population de base

2-Evaluer cette population

3-Selection de la population (tournoi)

4-Accoupler des individus pour donner naissance à des enfants

5-Faire muter certains individus

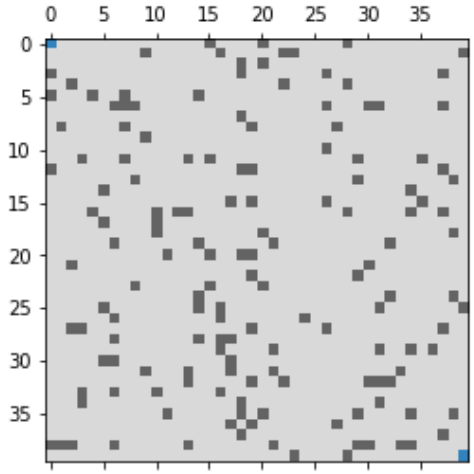
6-Recommencer

Génération de labyrinthe :

Le labyrinthe étant une matrice, on utilise une fonction qui génère une matrice de taille paramétrée avec 10% de 1 et 90% de 0 (avec un impératif sur les cases de départ et d'arrivée non nulles).

RENDU D'EXPERIENCE

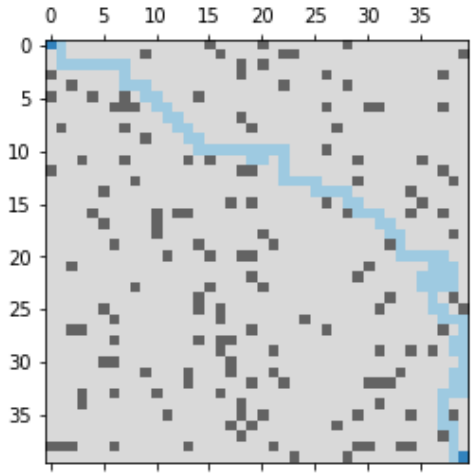
40 * 40



Entrée [5]:
```
# 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)]
```

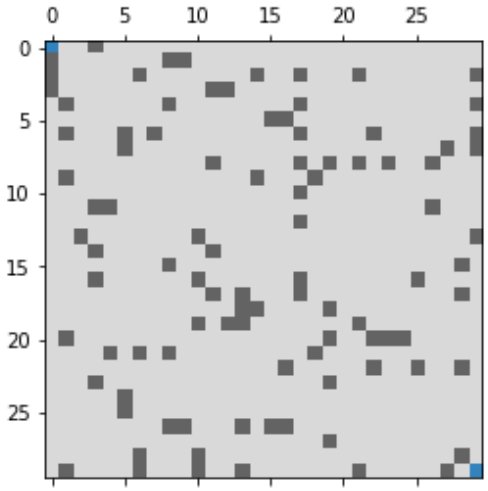
```
Dimension : 40x40
Work in progress...please wait
population generation number : 20
time elapsed : 10.5761137008667
```

Entrée [6]:
```
display_labyrinth(grid, start, end, solution)
```



30 * 30

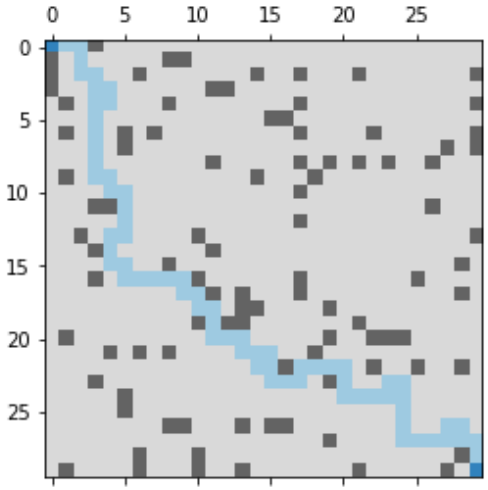
```
No solution has been found
```



trée [5]:
```
# 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)]
```

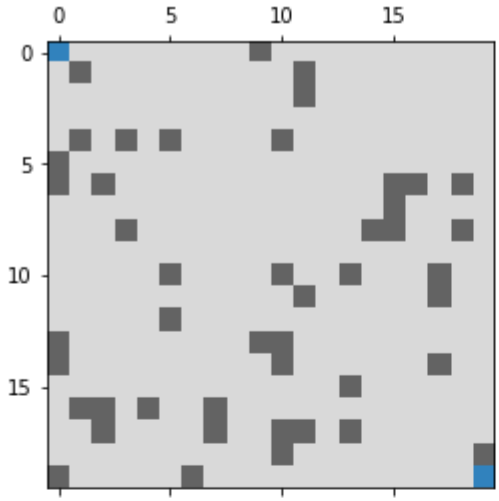
```
Dimension : 30x30
Work in progress...please wait
population generation number : 23
time elapsed : 5.39858865737915
```

trée [6]:
```
display_labyrinth(grid, start, end, solution)
```



*20 * 20*

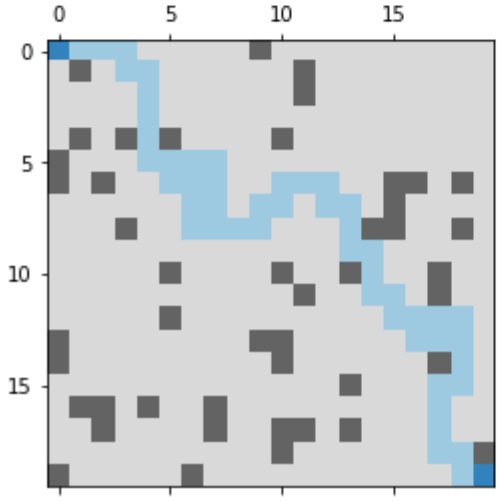
```
No solution has been found
```



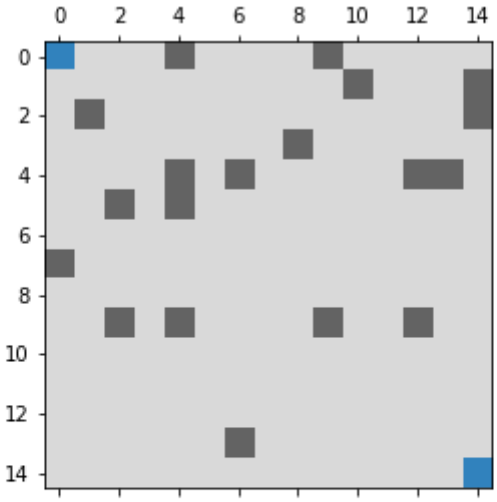
rée [5]:
```
# Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 10)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3), (2,3), (3,3), (4,3),
#             (5,3), (6,3), (6,4), (6,5), (6,6), (6,7), (6,8), (7,8),
#             (8,8), (9,8), (9,9)]
```

```
Dimension : 20x20
Work in progress...please wait
population generation number : 8
time elapsed : 0.9703474044799805
```

rée [6]:
```
display_labyrinth(grid, start, end, solution)
```



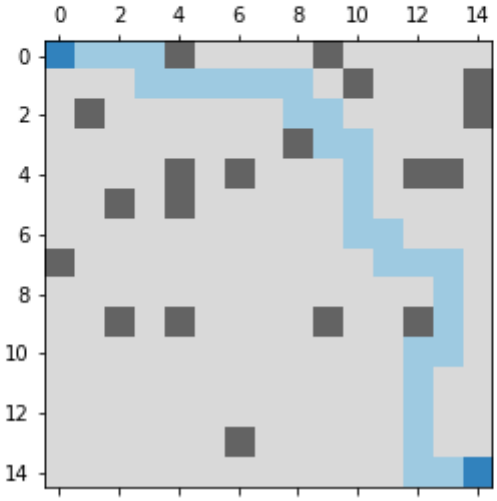
15 * 15



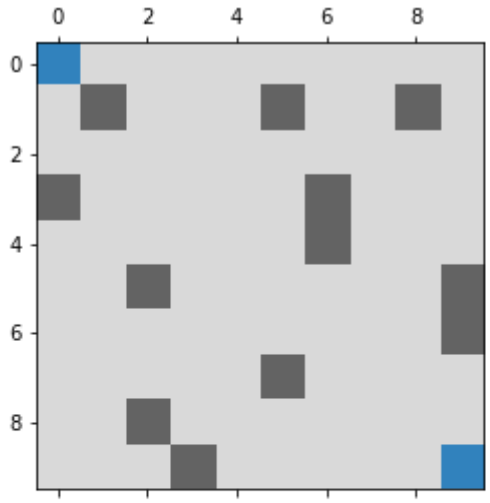
ıtrée [5]:
```
# Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 10)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3),
#             (5,3), (6,3), (6,4), (6,5), (6,6), (
#             (8,8), (9,8), (9,9)]
```

```
Dimension : 15x15
Work in progress...please wait
population generation number : 6
time elapsed : 0.32636404037475586
```

ıtrée [6]:
```
display_labyrinth(grid, start, end, solution)
```



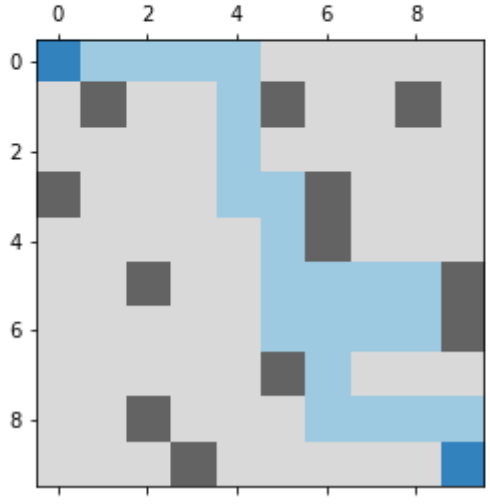
10 * 10



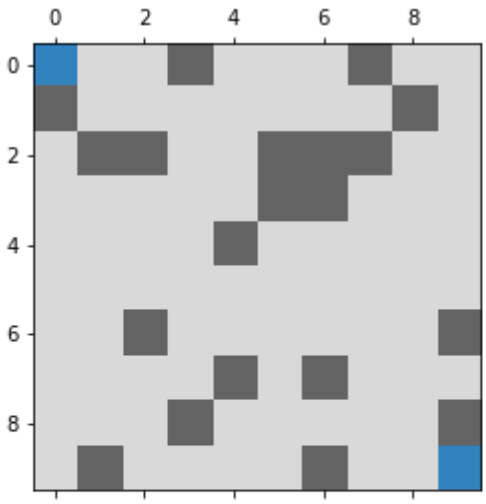

```
]: # Run the genetic algorithm
solution = solve_labyrinth(grid, start, end, 10)
# solution = [(0,0), (0,1), (0,2), (1,2), (1,3),
#             (5,3), (6,3), (6,4), (6,5), (6,6), (
#             (8,8), (9,8), (9,9)]
```

```
Dimension : 10x10
Work in progress...please wait
population generation number : 2
time elapsed : 0.042810678482055664
```

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



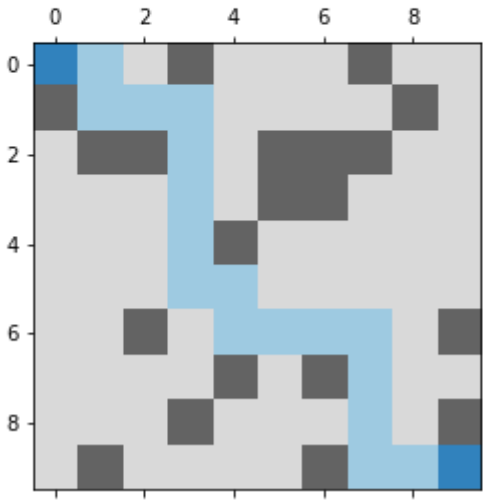
10 * 10 exemple




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

```
Dimension : 10x10
Work in progress...please wait
population generation number : 19
time elapsed : 0.3946402072906494
```

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



Conclusion

Au cours de ce TP nous avons pu mettre en pratique des connaissances théoriques apprises durant le cours d'intelligence artificielle concernant les algorithmes génétiques, nous avons ainsi pu prendre en main le Framework DEAP.

Les résultats présentés précédemment sont concordants avec l'aspect théorique.

En effet, il arrive que parfois l'expérience échoue, ou alors que le résultat soit instantané et échoue durant l'essai suivant.

Afin de prolonger ce TP, nous pourrions optimiser la fonction de fitness en la complexifiant afin d'être plus précis, ce qui permettrait alors de gommer certaines incohérences présentent dans certains chemins trouvés.

Sources

Voici des références qui ont été utiles lors de la réalisation de ce TP.

Tutoriel DEAP Python

T.P. Algorithmes GENETIQUES, Fabien Moutarde, Centre de Robotique (CAOR) MINES PARISTECH

Algorithmes Génétiques, 1998,Alain TISSERANT, MINES NANCY

TP/TD : Apprentissage, Evolution Artificielle et Algorithmes Genetiques, J.D. Zucker, Cours Master Université Pierre et Marie Curie