Marius Ballot

Introduction à la robotique - TP1 - Rapport

I. Étude de l'algorithme

1. Si ε = 1 ou ε = 0, on retrouverait l'algorithme de base de A* étant :
   f(nc) = g(nc) + h(nc, ngoal)

2. On perdrait l'admissibilité de la fonction h(nc, ngoal), car on surestimerait le coût minimum du chemin entre nc et ngoal,

II. Implémentation et test de l'algorithme A*

1. Cette partie du code est responsable du rendu de la ***Map***.

Je me suis permis d'optimiser le code en hydratant de manière dynamique les variables ***dimX*** et ***dimY*** avec les dimension de la matrice ***occupancyGrid*** directement.

```
332     occupancyGrid = np.array([
333     [0,0,0,0,0,0,0],
334     [0,1,1,0,1,1,0],
335     [0,0,1,0,0,1,0],
336     [1,0,0,0,0,1,0],
337     [0,1,1,1,0,1,0],
338     [0,0,0,0,0,1,0]])
339
340     # max adjacency degree
341     adjacency = 4
342
343     # dimensions of the map
344     dimX = occupancyGrid.shape[0]
345     dimY = occupancyGrid.shape[1]
```

2. La fonction ***manhattanDist***, et modification des fonctions ***heuristic*** et ***distance*** :

```
def heuristic(node, nodeGoal):
    return manhattanDist(node, nodeGoal)


def distance(node1, node2):
    return manhattanDist(node1, node2)
# --- end of Astar ------------------------------

# distances

def manhattanDist(node1, node2):
    distance = abs(node1.x - node2.x) + abs(node1.y - node2.y)
    return distance
```

Teste de l'algo A* avec node de départ (0,0) et node d'arrivée (5,0) avec 4 d'adjacence :

```
startNodeNo = 0
goalNodeNo = 5
closedList, successFlag = carte.AStarFindPath(
    startNodeNo, goalNodeNo, epsilon=4.0)

print(successFlag)
```

Le ***successFlag*** retourné est ***True***, l'algorithme a bien fonctionné.

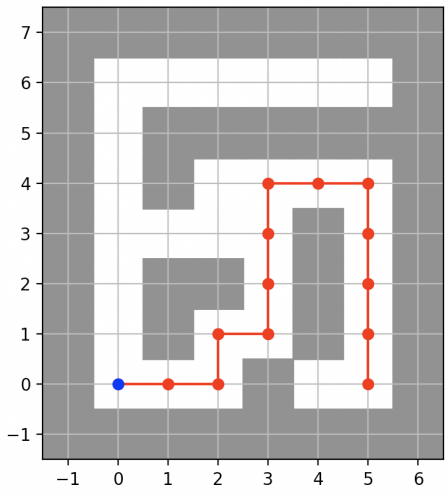
3. Voici ma méthode ***buildPath***. En reconstruisant le path depuis la fin de ***cList***, on peut reconstruire path via le ***parentNo*** de chaque node

```
def builtPath(self, closedList):
    # local copy to not modify closedList parameter => work wi
    cList = list(closedList)
    path = []

    path.insert(0, cList[-1])
    parentExits = True

    while parentExits:
        parNode = self.graph.listOfNodes[path[0]].parentNo

        if parNode == None:
            parentExits = False
        else:
            path.insert(0, parNode)
    return path
```

Voici d'ailleurs une comparaison de ***cList*** au dessus et ***Path*** en dessous.

```
[0, 1, 2, 8, 9, 15, 6, 21, 12, 20, 27, 28, 29, 23, 17, 11, 5]
[0, 1, 2, 8, 9, 15, 21, 27, 28, 29, 23, 17, 11, 5]
```

Voici la représentation du chemin en passant ***path*** dans ***plotPathOnMap()***



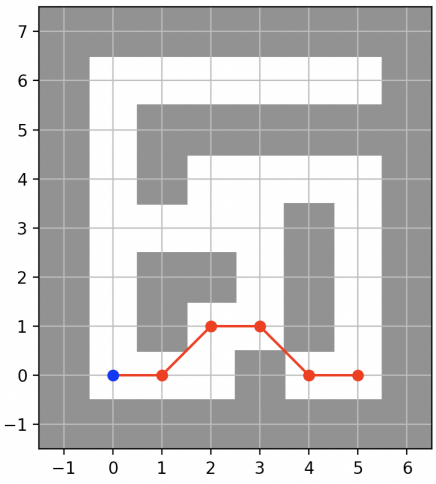
4. Voici le snippet de code permettant de faire le remplissage de la matrice d'adjacence avec un degré de 8.

```
if (self.graph.adjacency == 8):
    if (y-1 >= 0 and x-1 >= 0):  # down-left neighbour
        if (self.occupancy[x-1, y-1] == 0):
            self.graph.adjacencyMatrix[nc,
                                       nc-self.dimX-1] = 1

    if (y-1 >= 0 and x+1 < self.dimX):  # down-right neighbour
        if (self.occupancy[x+1, y-1] == 0):
            self.graph.adjacencyMatrix[nc,
                                       nc+1-self.dimX] = 1

    if (y+1 < self.dimY and x-1 >= 0):  # up-left neighbour
        if (self.occupancy[x-1, y+1] == 0):
            self.graph.adjacencyMatrix[nc,
                                       nc+self.dimX-1] = 1

    if (y+1 < self.dimY and x+1 < self.dimX):  # up-right neighbour
        if (self.occupancy[x+1, y+1] == 0):
            self.graph.adjacencyMatrix[nc,
                                       nc+1+self.dimX] = 1
```

5. Remplissage de la fonction ***euclidianDist***

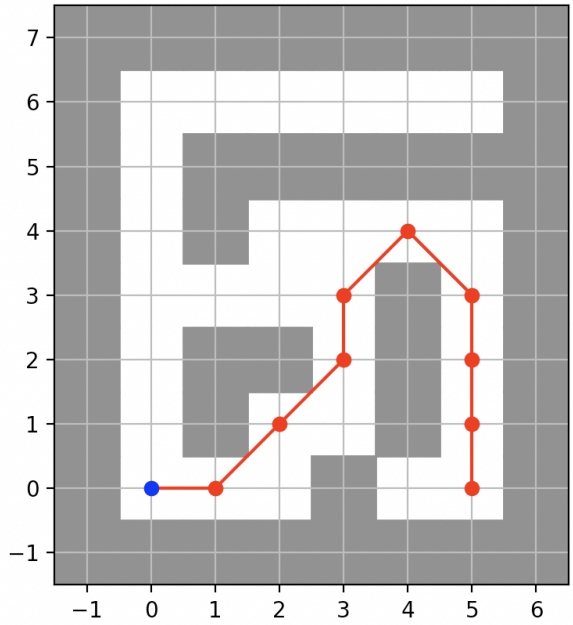
```
def euclidianDist(node1, node2):
    distance = math.sqrt(pow((node1.x-node2.x), 2) + pow((node1.y-node2.y), 2))
    return distance
```

Représentation du graph



Nous pouvons voir que ceci n'est pas le comportement attendue du fait que le chemin dessiner passe entre 2 obstacles en (3,0) et (4,1). Une solution est d'augmenter le coefficient ε.

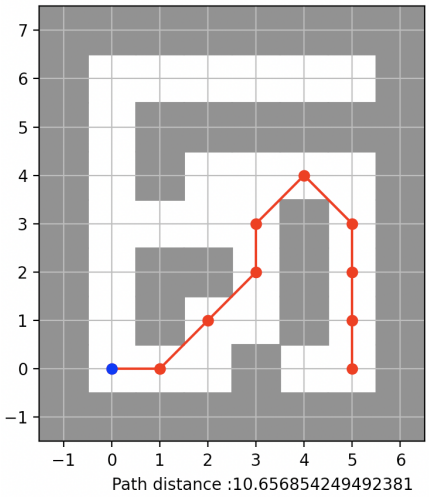
Ici avec ε=6, nous obtenons le comportement attendu.



III. Calcul de la distance parcourue:

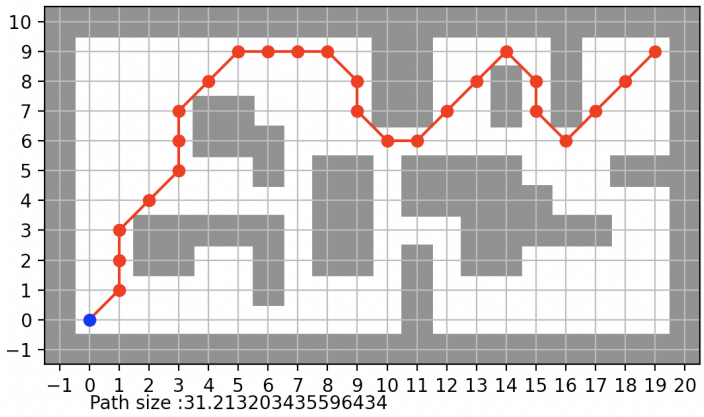
```python
def builtPath(self, closedList):
    # local copy to not modify closedList parameter => work with cList inside this function
    cList = list(closedList)
    distance = 0
    path = []

    path.insert(0, cList[-1])
    parentExits = True

    while parentExits:
        parNode = self.graph.listOfNodes[path[0]].parentNo

        if parNode == None:
            parentExits = False
        else:
            distance += euclidianDist(
                self.graph.listOfNodes[path[0]], self.graph.listOfNodes[parNode])
            path.insert(0, parNode)

    return path, distance
```

Affichage



Valeur de ε	0	1	2	3
Longueur du chemin	23.8	23.8	30.6	31.2
Nombre d'itérations	151	122	73	73
Nombre de nœuds explorés	152	187	32	28

Example de rendu pour ε = 3



On peut remarquer que plus ε est grand, moins le chemin est optimisé, mais le cout de recherche est aussi moins important