COMP4106 Assignment1
Technical Document

Name: Feifei Zhao Student ID: 101047476
Name: Evelyn Yang Student ID: 101099545
Name: Ruida Jiang Student ID: 101018602

ReadMe:

Purpose: For a given matrix(m x n), where m > 1, n > 1, with a valid start point ‘S’ and endpoint ‘G’, output the cost of the optimal path from ‘S’ to ‘G’ and write the optimal path and the explored coordinate in to file.

Source files:

1. main.py
2. aStar.py
3. map.py
4. pathfinding.py
5. SearchEntry.py

Input files:

1. input.csv (please change the input file to this name if different)

Output file:

1. optimal.txt (including optimal path)
2. explored.txt (including explored list)

Execution process:

1. python3 main.py (use “python3” if python2 and 3 version both exist, otherwise use “python” will work)

Comment:

Please make sure the test matrix is with dimension m>1 and n>1, otherwise, there’s no need to find the optimal path, so we decide not to handle it. The test format we used is exactly the same with sample input, the only difference is ending with .csv.

Description of the implementation

1. Used package: numpy

```
import numpy as np
```

2. Entry Setting:
   2.1 Read the input from the input file, convert it into a 2D array using numpy

```
my_data = np.genfromtxt('input.csv', dtype=str, delimiter=',',
      autostrip=True)
```

   2.2 Create a class to store the matrix we read from the input file (input.csv).

```
class Map():
    def __init__(self, width, height):
        self.width = width
        self.height = height
        self.ht = my_data
```

   2.3 Create a class to store the one step of the path finding process.

```
class SearchEntry():
    def __init__(self, x, y, g_cost, f_cost=0, pre_entry=None):
        self.x = x
        self.y = y
        self.g_cost = g_cost  # g
        self.f_cost = f_cost  # f
        self.pre_entry = pre_entry  # parent

    # current node position
    def getPos(self):
        return (self.x, self.y)
```

   2.4 Search for the start point and endpoint, they both read from a map, and startP will return the coordinate(tuple) and findE will return a list of coordinate(s) for desired position.

```
def startP():

    s = list(zip(*np.where(my_data == "S")))
    s1 = s[0]
    return s1

def findE():

    e = list(zip(*np.where(my_data == "G")))
    return e
```

3. A* Search Algorithm

Function definition:

```
def AStarSearch(map, source, dest, temp_explored):
```

3.1 We need to define the cost function **f(n)** for optimal path finding where:

f(n)= g(n) + h(n)

g(n): the cost for a specific node to move from its neighbors.
h(n): the heuristic function (estimated cost)

3.2 Define h(n) in calHeruistic():

```
def calHeuristic(pos, dest):
    return abs(dest.x - pos[0]) + abs(dest.y - pos[1])
```

3.3 Main body logic:

1. Find the frontier: use getPos() to create a list of coordinates that could be visited, we avoid adding the obstacles ('X' and 'G' if multiple goals) into the list.
2. Use addAdjacentPositions() to calculate the cost of getting to each point towards the goal state 'G'.
3. Use getFastPosition() to check which is the smaller f(n)
4. Find the optimal cost and create an openList dictionary to store the location calculated by searchEntry and a closeList dictionary to store the explored coordinates by using the result in step 3.
5. For each point (variable location), if it has a parent coordinate that moves to it, then store it as part of the optimal path.
6. we store the explored list into **temp_explored**
7. After finding the optimal path, we calculate the optimal cost.

4. Pathfinding

function:

```
def pathfinding(input_filename, optimal_path_filename,
explored_list_filename):
```

a. Read the input and create a map, illustrated in part 1.
b. Call startP and findE() to get the start point and endpoint, illustrated in part 2.

c. Since there might exist multiple possible destinations, then using a for loop to see which goal is the optimal one. If the previous cost is larger than current, replace the result with the current result.
d. Write the final optimal path into “optimal.txt”.
e. Write the final explored coordinates into “explored.txt”.

5. Main()
   Main function calls pathfinding on ‘input.csv’, this file needs to be existed with an accurate name, otherwise, this file name needs to be changed. Other two file names are the file that the optimal path and explored list will be written into, they can be changed to any name.

```
if __name__ == '__main__':
	optimalcost = pathfinding('input.csv','optimal.txt',
				'explored.txt')
	print("optimal cost is:",optimalcost)
```

Question:

1. What type of agent have you implemented?
   In our implementation we use **Utility-Based** agents to implement.

   Utility-Based agents act to optimize a utility function → f(n)
   Utility functions measures how close we are to optimizing the performance measure

   The node can move up, down, left, right, when meeting the wall/obstacle it will stop.

   Below function allows it to move to any of 4 directions.

```
def getPositions(map, location):
	offsets = [(-1, 0), (0, -1), (1, 0), (0, 1)]
	poslist = []
	for offset in offsets:
		pos = getNewPosition(map, location, offset)
		if pos is not None:
			flag = pos==dest.getPos()
			if(flag==True):
				ht[pos[0]][pos[1]] = 0
			if(ht[pos[0]][pos[1]]=='S'):
				ht[pos[0]][pos[1]]=0
			if(ht[pos[0]][pos[1]]!='X' and
					ht[pos[0]][pos[1]]!='G' ):
				poslist.append(pos)
```

Based on the theory of Utility-Based agents, it aims at agents which can consider multiple goals with some trade-off. After calculating the best path, our function can help find the path with lowest cost.

2. What is the best heuristic to use in A* search for this environment? Recall that there may be multiple goal states.

   A* heuristics is based on the function: f(n) = n.path.cost + h(n)

   dest.x the cost of the best path from x to y

   For movements are strictly constrained in for directions, the best estimation is using Manhattan heuristic function, which can be express as:

```
abs(dest.x - pos[0]) + abs(dest.y - pos[1])
```

3. Suggest a particular instance of this problem where A* search would find the optimal solution faster than uniform cost search.

```
1, 1, X
2, S, 2
3, 2, G
```

   Consider this map, if running A* search, then the explored list will be [(1, 1), (2, 1), (1, 2), (2, 2)], but if running uniform cost search, then the explored list will be [(1, 1), (0, 1), (1, 0), (2, 1), (1, 2), (2, 2)], thus A* search would find the optimal solution faster than uniform cost search.

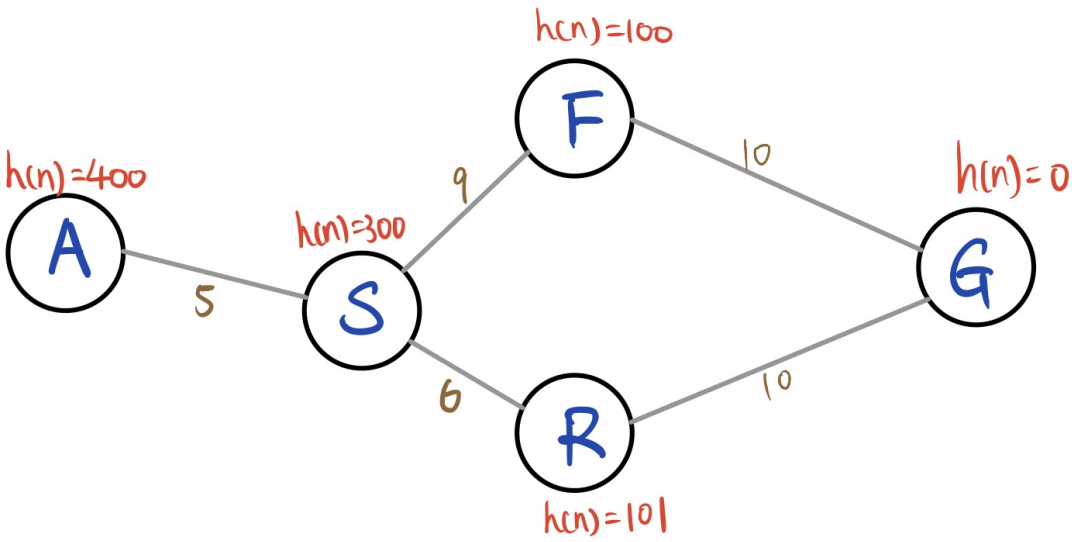
4. Suggest a particular instance of this problem where a greedy heuristic search would not find the optimal solution.

   When using greedy search to make the robot move we use f(n) = h(n)

   instead of f(n) = g(n) + h(n).

   the greedy search algorithm is **not complete**, that is, there is always the risk to take a path that does not bring to the goal, all nodes on the *border* (or fringe or frontier) are kept in

memory, and nodes that have already been expanded do not need to be stored in memory and can therefore be discarded. In general, the greedy search is also **not optimal**



In this picture, we can see the difference between the A* and the greedy heuristic. We're going from PointA to PointG, In this case, both A*heuristic and greedy heuristic choose pointS. For greedy heuristics will search pointF have the lowest f(n) = h(n) = 100. However A* will search pointR, f(n)=h(n)+g(n)=5+6+101=112. However, the real result of pointF should be f(n)=100+9+5=114. Thus it can be seen that the greedy heuristic cannot find the optimal path.

5. Consider a variant of this problem where the agent can move to adjacent diagonal squares. What would be the best heuristic to use in A* search if the agent could also make such diagonal moves?

   When nodes are allowed to make diagonal moves, by the property of triangle, the diagonal move may cost less than move along the other two sides.
   Hence the heuristic function by using Manhattan heuristic function may not be the best estimation. Now we can consider two other heuristic functions:

   1. Diagonal/Chebyshev heuristic:

      If diagonal cost is same as moving in 4 basic directions, then we could use:

      h(n) = Cost * max(abs(n.x-goal.x), abs(n.y-goal.y))

If they cost different, then we can combine them where straight moves using Manhattan heuristic function and diagonal using Chebyshev heuristic function to get a better estimation.

2. Euclidean:

   h(n) = D * sqrt((n.x-goal.x)^2 + (n.y-goal.y)^2)

   This will still give us the optimal path, but it will search more coordinates which increase the complexity

Statement of Contribution:

1. whether each group member made significant contribution

   Yes, everyone has made an outstanding contribution

2. whether each group member made an approximately equal contribution

   Yes

3. which aspects of the assignment each group member contributed to

   Feifeizhao:
   1. Implement A* search
   2. Debug
   3. Testing program
   4. Documentation

   Evelyn Yang：
   1. Implement A* search algorithm
   2. Debug
   3. Testing program
   4. Documentation

   Ruida Jiang:
   1. Research for implementation of A* search algorithm
   2. Research for the documentation questions
   3. Documentation
   4. Testing program