Junaid A Girkar
60004190057
TE Comps A4

ARTIFICIAL INTELLIGENCE
EXPERIMENT 3

Aim: Identify and analyze informed search algorithms to solve the problem.
Implement A* search algorithm to reach goal state.

Theory:
A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals.
Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections.
And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Explanation:
Consider a square grid having many obstacles and we are given a starting cell and a target cell. We want to reach the target cell (if possible) from the starting cell as quickly as possible. Here A* Search Algorithm comes to the rescue.
What A* Search Algorithm does is that at each step it picks the node according to a value-‘f’ which is a parameter equal to the sum of two other parameters – ‘g’ and ‘h’. At each step it picks the node/cell having the lowest ‘f’, and process that node/cell.

We define ‘g’ and ‘h’ as simply as possible below:
g = the movement cost to move from the starting point to a given square on the grid, following the path generated to get there.
h = the estimated movement cost to move from that given square on the grid to the final destination. This is often referred to as the heuristic, which is nothing but a kind of smart guess. We really don’t know the actual distance until we find the path, because all sorts of things can be in the way (walls, water, etc.). There can be many ways to calculate this ‘h’.

Heuristics:
A) Either calculate the exact value of h (which is certainly time consuming).
OR
B) Approximate the value of h using some heuristics (less time consuming).

We will discuss both of the methods.
A) Exact Heuristics –
We can find exact values of h, but that is generally very time consuming.

Below are some of the methods to calculate the exact value of h.

1) Pre-compute the distance between each pair of cells before running the A* Search Algorithm.
2) If there are no blocked cells/obstacles then we can just find the exact value of h without any pre-computation using the distance formula/Euclidean Distance

B) Approximation Heuristics –
There are generally three approximation heuristics to calculate h –
1) Manhattan Distance –

- It is nothing but the sum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
h = abs (current_cell.x - goal.x) + (current_cell.y - goal.y)
```

- When to use this heuristic? – When we are allowed to move only in four directions only (right, left, top, bottom)

2) Diagonal Distance-

- It is nothing but the maximum of absolute values of differences in the goal's x and y coordinates and the current cell's x and y coordinates respectively, i.e.,

```
dx = abs(current_cell.x - goal.x)
dy = abs(current_cell.y - goal.y)

h = D * (dx + dy) + (D2 - 2 * D) * min(dx, dy)
```

where D is the length of each node(usually = 1) and D2 is diagonal distance between each node (usually = sqrt(2)).

- When to use this heuristic? – When we are allowed to move in eight directions only (similar to a move of a King in Chess)

3) Euclidean Distance-

- As it is clear from its name, it is nothing but the distance between the current cell and the goal cell using the distance formula

```
h = sqrt ((current_cell.x - goal.x)2 + current_cell.y - goal.y)2)
```

- When to use this heuristic? – When we are allowed to move in any directions.

Limitations

Although being the best pathfinding algorithm around, A* Search Algorithm doesn't produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h

Time Complexity

Considering a graph, it may take us to travel all the edge to reach the destination cell from the source cell [For example, consider a graph where source and destination nodes are connected by a series of edges, like – 0(source) –>1 –> 2 –> 3 (target)
So the worst case time complexity is O(E), where E is the number of edges in the graph

Algorithm:

```
// A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search
          successor.g = q.g + distance between
                              successor and q
          successor.h = distance from goal to
          successor (This can be done using many
          ways, we will discuss three heuristics-
          Manhattan, Diagonal and Euclidean
          Heuristics)

          successor.f = successor.g + successor.h

        ii) if a node with the same position as
            successor is in the OPEN list which has a
           lower f than successor, skip this successor

        iii) if a node with the same position as
            successor  is in the CLOSED list which has
            a lower f than successor, skip this successor
            otherwise, add  the node to the open list
     end (for loop)

    e) push q on the closed list
    end (while loop)
```

Code:

```
from collections import deque

class Graph:
    def __init__(self, adjac_lis):
        self.adjac_lis = adjac_lis

    def get_neighbors(self, v):
        return self.adjac_lis[v]

    def heuristic(self, n):
        H = {
            'S': 15,
            '1': 14,
            '2': 10,
            '3':  8,
            '4': 12,
            '5': 10,
            '6': 10,
            '7': 0
        }

        return H[n]

    def a_star(self, start, stop):

        open_list = set([start])
        closed_list = set([])

        distance = {} # Distance from Start.
        distance[start] = 0

        adjacent_nodes = {} # Adjacent Mapping of all Nodes
        adjacent_nodes[start] = start

        while len(open_list) > 0:
            n = None

            for v in open_list:
                if n == None or distance[v] + self.heuristic(v) < distance[n] + self.heuristic(n):
                    n = v

            if n == None:
                print('Path does not exist!')
                return None
            if n == stop: # If current node is stop, we restart
                reconst_path = []

                while adjacent_nodes[n] != n:
                    reconst_path.append(n)
                    n = adjacent_nodes[n]

                reconst_path.append(start)

                reconst_path.reverse()

                print('\nPath found: {}\n'.format(reconst_path))
                return reconst_path

            for (m, weight) in self.get_neighbors(n): # Neighbours of current node

                if m not in open_list and m not in closed_list:
                    open_list.add(m)
                    adjacent_nodes[m] = n
                    distance[m] = distance[n] + weight

                else: # Check if its quicker to visit n then m
                    if distance[m] > distance[n] + weight:
                        distance[m] = distance[n] + weight
                        adjacent_nodes[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)


            open_list.remove(n) # Since all neighbours are inspected
            closed_list.add(n)
            print("OPEN LIST : ", end="")
            print(open_list)
            print("CLOSED LIST : ", end="")
            print(closed_list)
            print("-----------------------------------")

        print('Path does not exist!')
        return None


adjacent_list2 = {
        'S': [('1', 3), ('4', 4)],
        '1': [('S', 3), ('2', 4), ('4', 5)],
        '2': [('1', 4), ('3', 4), ('5', 5)],
        '3': [('2', 4)],
        '4': [('S', 4), ('1', 5), ('5', 2)],
        '5': [('4', 2), ('2', 5), ('6', 4)],
        '6': [('5', 4), ('7', 3)],
        '7': [('6', 3)],
}

g = Graph(adjacent_list2)
g.a_star('S', '7')
```

Output:

```
OPEN LIST : {'1', '4'}
CLOSED LIST : {'S'}
----------------------------------
OPEN LIST : {'5', '1'}
CLOSED LIST : {'S', '4'}
----------------------------------
OPEN LIST : {'2', '6', '1'}
CLOSED LIST : {'S', '5', '4'}
----------------------------------
OPEN LIST : {'2', '6'}
CLOSED LIST : {'S', '1', '5', '4'}
----------------------------------
OPEN LIST : {'6', '3'}
CLOSED LIST : {'2', '5', 'S', '1', '4'}
----------------------------------
OPEN LIST : {'6'}
CLOSED LIST : {'2', '5', 'S', '3', '1', '4'}
----------------------------------
OPEN LIST : {'7'}
CLOSED LIST : {'2', '5', '6', 'S', '3', '1', '4'}
----------------------------------


Path found: ['S', '4', '5', '6', '7']
```

Conclusion: We learnt about A star algorithm, its uses, limitations and implemented it in a python code.