Shivam Tiwari
5117060
BE comps

Experiment 02

- **Aim:-** Write a program to implement A* Algorithm.

- **Theory:-**

Informed Search (Heuristic Search): This can decide whether one non-goal state is more promising than another non-goal state.

- **The A* Algorithm.**

(1) A* is the most popular form of best first search.
(2) A* evaluates nodes based on two functions. namely.
(1) $g(n)$ – The cost to reach the node 'n'.
(2) $h(n)$ – The cost to reach the goal node from node 'n'.
These two function's cost are combined into one, to evaluate a node New function $f(n)$ is derived as.

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

$f(n)$ = Estimated cost of cheapest solution through n.

- **Working of A***

1) The algorithm maintains two sets.
a) Open list → nodes to be examined
b) Closed list → Already been examined.
2) Initially the open list contains just initial node & closed list is empty. Each node n contains. maintains the following: $g(n)$, $h(n)$, $f(n)$
3) Each node also maintains a pointer to its parent so that later the best solution, if found can be retrieved. A* has a main loop that represents repeatedly get the node, call it 'n', with the lowest $f(n)$. value from the OPEN list. If 'n' is the goal node, then we are done and solution is giving by backtracking from n'.

4. for each successor node of 'n', if it is already in the CLOSED list & the copy there has an equal or lower 'f' estimate & then we can safely discard the newly generated 'n' & move on.
Similarly if 'n' is already in the OPEN list and the copy there has an equal or lower f estimate we can discard the newly generated 'n' & move on.
5. If no better version of 'n' exists on either the CLOSED or OPEN List & we remove the inferior copies from the two list & set 'n' as the parent of 'n'. We also calculate the cost estimate for 'n' as follows :- set g(n) which is g(n) plus cost of getting n from set h(n) is the heuristic estimate of getting from n to the goal node
set f(n) is g(n) + h(n)
6. lastly add 'n' to the OPEN list & return to the begining of the main loop.

- Performance Measure for A*
1. Completeness : A* is complete & guarantees soln.
2. Optimality :- A* is optimal if h(n) never overestimates the cost to reach the goal node. It is ~~consisten~~ optimal if h(n) is consistent
3. time & Space Complexity :-
time increases as the number of nodes to reach goal node increase
A* has a problem of space as it stores all generated nodes & it runs out of memory before time.

- Conclusion :- Thus we have implemented & studied A* search Algorithm along with its performance measure.

Rollno: 5117060

A* SEARCH

CODE:

```
class Node():
    """A node class for A* Pathfinding"""
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
        self.g = 0
        self.h = 0
        self.f = 0
    def __eq__(self, other):
        return self.position == other.position
def astar(maze, start, end):
    """Returns a list of tuples as a path from the given start
to the given end in the given maze"""
    # Create start and end node
    start_node = Node(None, start)
    start_node.g = start_node.h = start_node.f = 0
    end_node = Node(None, end)
    end_node.g = end_node.h = end_node.f = 0
    # Initialize both open and closed list
    open_list = []
    closed_list = []
    # Add the start node
    open_list.append(start_node)
    # Loop until you find the end
    while len(open_list) > 0:
        # Get the current node
        current_node = open_list[0]
        current_index = 0
        for index, item in enumerate(open_list):
            if item.f < current_node.f:
                current_node = item
                current_index = index
        # Pop current off open list, add to closed list
        open_list.pop(current_index)
        closed_list.append(current_node)
        # Found the goal
        if current_node == end_node:
            path = []
            current = current_node
            while current is not None:
                path.append(current.position)
                current = current.parent
            return path[::-1] # Return reversed path
        # Generate children
        children = []
        for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0),
(-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares
```

```
            # Get node position
            node_position = (current_node.position[0] +
new_position[0], current_node.position[1] + new_position[1])
            # Make sure within range
            if node_position[0] > (len(maze) - 1) or
node_position[0] < 0 or node_position[1] >
(len(maze[len(maze)-1]) -1) or node_position[1] < 0:
                continue
            # Make sure walkable terrain
            if maze[node_position[0]][node_position[1]] != 0:
                continue
            # Create new node
            new_node = Node(current_node, node_position)
            # Append
            children.append(new_node)
        # Loop through children
        for child in children:
            # Child is on the closed list
            for closed_child in closed_list:
                if child == closed_child:
                    continue
            # Create the f, g, and h values
            child.g = current_node.g + 1
            child.h = ((child.position[0] -
end_node.position[0]) ** 2) + ((child.position[1] -
end_node.position[1]) ** 2)
            child.f = child.g + child.h
            # Child is already in the open list
            for open_node in open_list:
                if child == open_node and child.g >
open_node.g:
                    continue
            # Add the child to the open list
            open_list.append(child)
if __name__ == '__main__':
    maze = [[0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
            [0, 0, 0, 0, 0, 0, 0, 0, 0, 0]]
    print("Maze is")
    for i in range(len(maze)):
        print(maze[i])
    start = (0, 0)
    end = (7, 6)
    print("Start at:",start)
```

```
    print("End at:", end)
    path = astar(maze, start, end)
    print("Path is:",path)
```

OUTPUT:

```
OUTPUT:
 Maze is
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
Start at: (0, 0)
End at: (7, 6)
Path is: [(0, 0), (1, 1), (2, 2), (3, 3), (4, 3), (5, 4), (6, 5), (7, 6)]
```