Name: Sakshi Jadhav
Class: BE IT B
Roll: 49
Batch: B3

EXPERIMENT NO.4

Aim: To study Informed search strategy A* algorithm.

Theory:

Informed (Heuristic) Search Strategies

To solve large problems with large numbers of possible states, problem-specific knowledge needs to b e added to increase the efficiency of search algorithms.

Heuristic Evaluation Functions

They calculate the cost of the optimal path between two states. A heuristic function for sliding-tiles games is computed by counting the number of moves that each tile makes from its goal state and adding these number of moves for all tiles.

Pure Heuristic Search

It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes.

In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open list according to their heuristic value. The shorter paths are saved and the longer ones are disposed of.

A * Search

It is the best-known form of Best First search. It avoids expanding paths that are already expensive, but expands most promising paths first.

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

- g(n) the cost (so far) to reach the node
- h(n) estimated cost to get from the node to the goal
- f(n) estimated total cost of path through n to goal. It is implemented using priority queue by increasing f(n).

The A* algorithm is the best known Best First Search which is a type of search that utilizes a heuristic (an admissible heuristic is one that never **overestimates** the cost to go to the end from that point) in order to decide in which way a path should be expanded. To calculate the heuristic we usually do distance so far + heuristic = priority. Whichever point has the lowest priority value is the node that is traversed to the next.

Typically, the A* algorithm is typically used for graphs and graph traversals. In terms of graphs, A* is used for finding the shortest path to a certain point from a given point. This can be extended to the real world, it is used for routing. I believe that Google Maps and other such routing services use the A* algorithm in order to find the path you should take to minimize your time on the road. In that case, the heuristic would usually be the physical distance * some traffic factors. The algorithm can be used for various other spin-offs of this problem, such as games where you want to find the shortest way through a maze or "word ladder games"

A* is basically the way we can route so quickly. By avoiding poor decisions, we can save time (and memory) and also take the correct path.

Algorithm:

Step 1: Create a single member queue consisting of a root node.
Step 2: If the first member of the queue is the goal node then go to step 5
Step 3: If the first member is not the goal node then add it to the list of visited nodes and consider its children if any which are not explored. For each and every child node calculate the evaluation function which is f(n)=g(n)+h(n) where g(n) is the cost of path from the start node to another node n and h(n) is the estimated cost of the cheapest path to reach the goal node from

goal node n. From the expanded child nodes expand the node with the lowest evaluation function and add its parent node to the list of visited nodes.

Search the entire tree in the similar way until the goal node is achieved with the lowest evaluation function.

Step 4: If the queue is not empty then go to step number 2 else go to step number 6.

Step 5: Print success & stop.

Step 6: Print failure & stop.

Code:

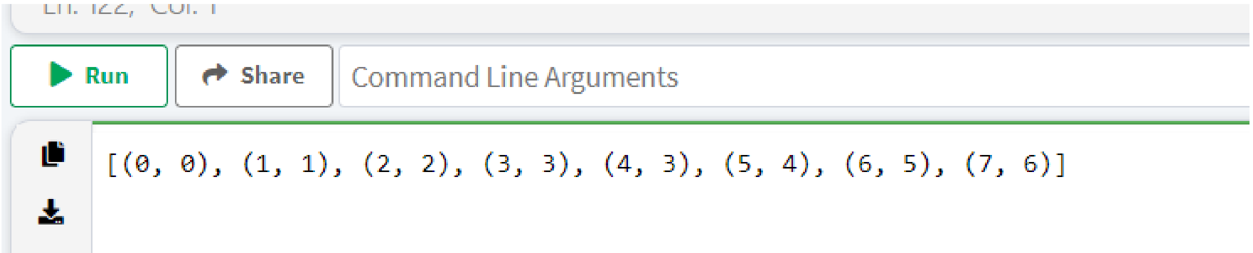
```
class Node():
    """A node class for A* Pathfinding"""

    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)


def 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]]

    start = (0, 0)
    end = (7, 6)

    path = astar(maze, start, end)
    print(path)
```

```
if __name__ == '__main__':
    main()
```

Output:-



Conclusion: Thus studied A star search .