Name: Kartik Jolapara   **SAPID:** 60004200107
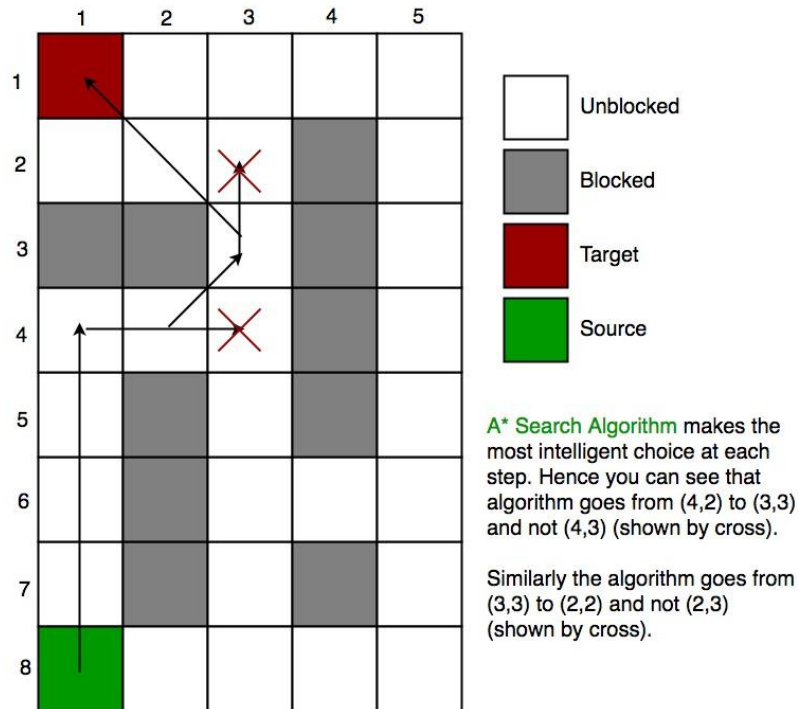
DIV: B/B1

A.I.

Exp3

Aim: To study and implement A* Algorithm

Theory:

It is a searching algorithm that is used to find the shortest path between an initial and a final point. It is a handy algorithm that is often used for map traversal to find the shortest path to be taken. A* was initially designed as a graph traversal problem, to help build a robot that can find its own course. It still remains a widely popular algorithm for graph traversal.

It searches for shorter paths first, thus making it an optimal and complete algorithm. An optimal algorithm will find the least cost outcome for a problem, while a complete algorithm finds all the possible outcomes of a problem. Another aspect that makes A* so powerful is the use of weighted graphs in its implementation. A weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost, and find the best route in terms of distance and time.

A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal path between two nodes in a graph. It will be used for the shortest path finding. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

| | 1 | 2 | 3 | 4 | 5 |

Unblocked

Blocked

Target

Source

A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).

Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Algorithm:

The open list must be initialized.

Put the starting node on the open list (leave its f at zero). Initialize the closed list.

Follow the steps until the open list is non-empty:

Find the node with the least f on the open list and name it "q".

Remove Q from the open list.

Produce q's eight descendants and set q as their parent.

For every descendant:

i) If finding a successor is the goal, cease looking ii)Else, calculate g and h

for the successor. successor.g = q.g + the calculated distance between the successor and the q.

successor.h = the calculated distance between the successor and the goal. We will cover three heuristics to do this: the Diagonal, the Euclidean, and the Manhattan heuristics.

successor.f = successor.g plus successor.h

iii)   Skip this successor if a node in the OPEN list with the same location as it but a lower f value than the successor is present.

iv)   Skip the successor if there is a node in the CLOSED list with the same position as the successor but a lower f value; otherwise, add the node to the open list end (for loop).

Push Q into the closed list and end the while loop.

Code:

```
def aStarAlgo(start_node, stop_node):

    open_set = set(start_node)

closed_set = set()        g = {}

parents = {}        g[start_node]

= 0

    parents[start_node] = start_node

while len(open_set) > 0:

        n = None          for v in open_set:          if n == None

or g[v] + heuristic(v) < g[n] + heuristic(n):          n = v


        if n == stop_node or Graph_nodes[n] == None:

pass          else:          for (m, weight) in

get_neighbors(n):          if m not in open_set and

m not in closed_set:

            open_set.add(m)

parents[m] = n          g[m]

= g[n] + weight          else:


            if g[m] > g[n] + weight:

g[m] = g[n] + weight

parents[m] = n          if m in

closed_set:
```

```python
                closed_set.remove(m)
                open_set.add(m)


        if n == None:
            print('Path does not exist!')
            return None            if n ==
            stop_node:            path = []
            while parents[n] != n:
            path.append(n)                n =
            parents[n]


                path.append(start_node)
            path.reverse()            print('Path found:
            {}'.format(path))            return path


            closed_set.add(n)
        print('Path does not exist!')
        return None


def get_neighbors(v):    if
v in Graph_nodes:
    return Graph_nodes[v]
else:
        return None


def heuristic(n):
    H_dist = {
        'A': 11,
        'B': 6,
        'C': 99,
```

```
        'D': 1,
        'E': 7,
        'G': 0,
    }
    return H_dist[n]
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
}
aStarAlgo('A', 'G')
```

Output:


```
Path found: ['A', 'E', 'D', 'G']
```

Conclusion:

Thus, we successfully studied and implemented A* Algorithm