**EXPERIMENT NO : 05 DATE : 12/03/24**

**Aim**: Implement the A\* Search 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.

**Algorithm:**

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. In the event that we have a grid with many obstacles and we want to get somewhere as rapidly as possible, the A\* Search Algorithms are our saviour. From a given starting cell, we can get to the target cell as quickly as possible , it is the sum of two variables’ values that determines the node it picks at any point in time. At each step, it picks the node with the smallest value of ‘f’ (the sum of ‘g’ and ‘h’) and processes that node/cell.

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

g’ and ‘h’ is defined as simply as possible below:

‘g’ is the distance it takes to get to a certain square on the grid from the starting point, following the path we generated to get there.

‘h’ is the heuristic, which is the estimation of the distance it takes to get to the finish line from that square on the grid.

Procedure A\*()

Open ← List(start)

f(start) ← h(start)

parent (start) ← NIL

closed ← {}

while open is not EMPTY

do

Remove node n from open such that f(n) has the lowest value

Add n to closed

if GoalTest (n) = TRUE

then return ReconstructPath(n)

neighbours - MoveGen (n)

for each m ∈ neighbours

do switch

case open AND closed: / new node \*/

Add m to open

parent (m) ← n

g(m) ← g(n) + k(n, m)

f(m) ← g(m) +h(m)

case m ∈ open :

if (g(n) +k(n,m)) < g(m)

then

parent(m) ← n

g(m) ← g(n) + k(n, m)

f(m) ← g(m) +h(m)

case m ∈ closed :

if (g(n) +k(n,m)) < g(m)

then

parent(m) ← n

g(m) ← g(n) + k(n, m)

f(m) ← g(m) +h(m)

PropagateImprovement (m)

return FAILURE

PropagateImprovement (m)

neighbours ← MoveGen (m)

for each s ∈ neighbours

do

newGvalue ← g(m) + k(m, s)

if newGvalue < g(s)

then

parent(s) ← m g(s) + newGvalue

if s ∈ closed

then

PropagateImprovement(s)

**Code:**

def Astar(graph, heuristic):

Snode = input("Enter the starting node: ")

Enode = input("Enter the ending node: ")

# open(node, parent, distance from goal, heuristic, total cost)

open\_list = [(Snode, None, 0, heuristic[Snode], 0 + heuristic[Snode])]

closed = []

parent = {Snode: None}

traversal = []

while len(open\_list) > 0:

state = open\_list.pop(0)

traversal.append(state)

temp = [(x, state[0], state[2] + graph[state[0]][x], heuristic[x], state[2] + graph[state[0]][x] + heuristic[x]) for x in graph[state[0]].keys() if x not in closed]

# Assign parent for new nodes

for x in temp:

parent[x[0]] = state[0]

if state[0] == Enode: # end node is found

break

# Insert into open list and sort it based on cost+heuristic

open\_list = temp + open\_list

open\_list = sorted(open\_list, key=lambda x: x[4])

# Insert into closed list

closed.append(state[0])

# Reconstruct path

current = Enode

path = [Enode]

while current != Snode:

path.insert(0, parent[current])

current = parent[current]

print("Traversal: ", end="")

prev = None

for node in traversal:

if node[0] in path and node[1] is prev:

print(node, end=" ")

prev = node[0]

# Print path

print("\n\nPath: ", end="")

for i in path:

print(i, end="\n" if i == Enode else "->")

graph = {

'A': {'B': 11, 'D': 32, 'H': 36},

'B': {'A': 11, 'C': 24, 'K': 42},

'C': {'B': 24, 'E': 40},

'D': {'S': 25, 'A': 32, 'F': 24, 'I': 26},

'E': {'C': 40, 'K': 32},

'F': {'D': 24, 'L': 27},

'H': {'J': 22, 'A': 36, 'K': 28, 'N': 44},

'G': {'T': 32, 'N': 42},

'I': {'D': 26, 'L': 21, 'M': 32},

'J': {'M': 20, 'H': 22},

'K': {'B': 42, 'E': 32, 'H': 28, 'N': 27, 'Q': 62},

'L': {'F': 27, 'O': 26, 'I': 21},

'M': {'I': 32, 'P': 23, 'J': 20},

'N': {'H': 44, 'K': 27, 'Q': 32, 'G': 42},

'O': {'L': 26, 'R': 27},

'P': {'M': 23},

'Q': {'K': 62, 'N': 32},

'R': {'O': 27, 'T': 52},

'S': {'D': 25},

'T': {'R': 52, 'G': 32}

}

# distance from goal

heuristic = {

'A': 11, 'B': 10, 'C': 8, 'D': 12, 'E': 5, 'F': 12, 'G': 0, 'H': 8, 'I': 10, 'J': 8, 'K': 6, 'L': 10, 'M': 7, 'N': 4, 'O': 8,

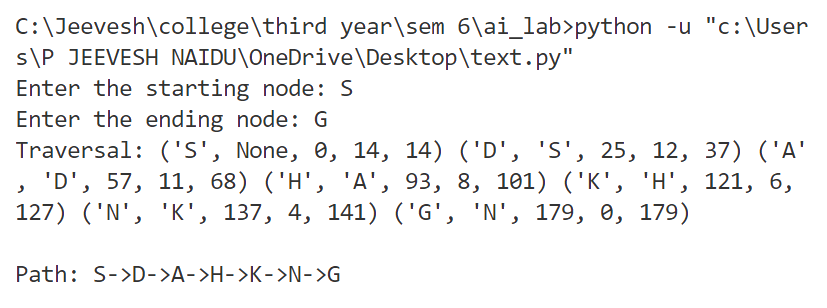
'P': 5, 'Q': 1, 'R': 6, 'S': 14, 'T': 2

}

# Call the A\* algorithm function

Astar(graph, heuristic

**Output:**

****

**Conclusion:**

Implementation of the A\* Search Algorithm was carried out by tracing the algorithm and above output was obtained during the same.