MIT Art Design and Technology University's
MIT School of Computing, Pune
Department of Computer Science and Engineering
BTech Third Year **A.Y.2023-24**

MIT-ADT UNIVERSITY
PUNE, INDIA
A leap towards World Class Education
(Established by MIT Art, Design and Technology University Act, 2015
(Maharashtra Act No. XXXIX of 2015)

Artificial Intelligence and Machine Learning Lab

Enrolment No: 03	**Division:** TY-CC2[B]
Roll No:2213774	**Name: Aditi Dhepe**

Experiment No.: 02

Title: Write a program to implement A* Algorithm

Theory:	A* search algorithm is an algorithm that separates it from other traversal techniques. This makes A* smart and pushes it much ahead of conventional algorithms. Let's try to understand Basic AI Concepts and comprehend how does A* algorithm work. Imagine a huge maze that is too big that it takes hours to reach the endpoint manually. Once you complete it on foot, you need to go for another one. This implies that you would end up investing a lot of time and effort to find the possible paths in this maze. Now, you want to make it less time-consuming. To make it easier, we will consider this maze as a search problem and will try to apply it to other possible mazes we might encounter in due course, provided they follow the same structure and rules. As the first step to converting this maze into a search problem, we need to define these six things. 1. A set of prospective states we might be in 2. A beginning and end state 3. A way to decide if we've reached the endpoint 4. A set of actions in case of possible direction/path changes 5. A function that advises us about the result of an action 6. A set of costs incurring in different states/paths of movement A* Search Algorithm Steps **Step 1: Add the beginning node to the open list** **Step 2: Repeat the following step** In the open list, find the square with the lowest F cost, which denotes the current square. Now we move to the closed square.

Consider 8 squares adjacent to the current square and Ignore it if it is on the closed list or if it is not workable. Do the following if it is workable.

Check if it is on the open list; if not, add it. You need to make the current square as this square's a parent. You will now record the different costs of the square, like the F, G, and H costs.

If it is on the open list, use G cost to measure the better path. The lower the G cost, the better the path. If this path is better, make the current square as the parent square. Now you need to recalculate the other scores – the G and F scores of this square.

– **You'll stop:**

If you find the path, you need to check the closed list and add the target square to it.

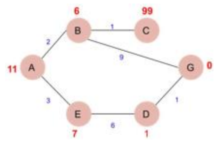
There is no path if the open list is empty and you cannot find the target square.

Step 3. Now you can save the path and work backward, starting from the target square, going to the parent square from each square you go, till it takes you to the starting square. You've found your path now.

Flowchart:

(if any)

In this section, we are going to find out how the A* search algorithm can be used to find the most cost-effective path in a graph. Consider the following graph below.



The numbers written on edges represent the distance between the nodes, while the numbers written on nodes represent the heuristic values. Let us find the most cost-effective path to reach from start state A to final state G using the A* Algorithm.

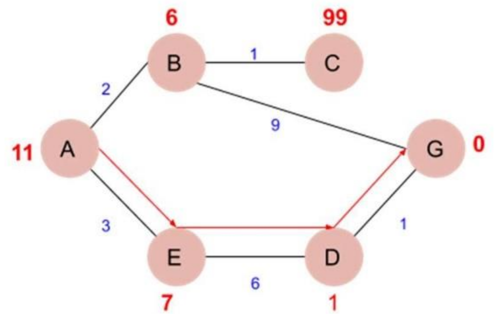
Let's start with node A. Since A is a starting node, therefore, the value of g(x) for A is zero, and from the graph, we get the heuristic value of A is 11, therefore

```
g(x) + h(x) = f(x)

0+ 11 =11

Thus for A, we can write

A=11

Now from A, we can go to point B or point E, so we compute
f(x) for each of them

A → B = 2 + 6 = 8

A → E = 3 + 6 = 9

Since the cost for  A → B is less, we move forward with this
path and compute the f(x) for the children nodes of B

Since there is no path between C and G, the heuristic cost is
set to infinity or a very high value

A → B → C = (2 + 1) + 99= 102

A → B → G = (2 + 9 ) + 0 = 11

Here the path A → B → G has the least cost but it is still
more than the cost of A → E, thus we explore this path
further

A → E → D = (3 + 6) + 1 = 10

Comparing the cost of A → E → D with all the paths we got so
far and as this cost is least of all we move forward with
this path. And compute the f(x) for the children of D

A → E → D → G = (3 + 6 + 1) +0 =10

Now comparing all the paths that lead us to the goal, we
conclude that A → E → D → G is the most cost-effective path
to get from A to G.
```

MIT Art Design and Technology University's
MIT School of Computing, Pune
Department of Computer Science and Engineering
BTech Third Year **A.Y.2023-24**

(Established by MIT Art, Design and Technology University Act, 2015 (Maharashtra Act No. XXXIX of 2015)

Artificial Intelligence and Machine Learning Lab



Next, we write a program in Python that can find the most cost-effective path by using the a-star algorithm.

First, we create two sets, viz- open and close. The open contains the nodes that have been visited, but their neighbours are yet to be explored. On the other hand, close contains nodes that, along with their neighbours, have been visited.

Pseudo code or Program

```
def aStarAlgo(start_node, stop_node):

        open_set = set(start_node)
        closed_set = set()
        g = {} #store distance from starting node
        parents = {}# parents contains an adjacency map of
all nodes

        #ditance of starting node from itself is zero
        g[start_node] = 0
        #start_node is root node i.e it has no parent
nodes
        #so start_node is set to its own parent node
        parents[start_node] = start_node


        while len(open_set) > 0:
            n = None

            #node with lowest f() is found
            for v in open_set:
                if n == None or g[v] + heuristic(v) < g[n]
+ heuristic(n):
                    n = v
```

This lab manual is prepared only for the students understanding and not for commercialization

```
            if n == stop_node or Graph_nodes[n] == None:
                pass
            else:
                for (m, weight) in get_neighbors(n):
                    #nodes 'm' not in first and last set
are added to first
                    #n is set its parent
                    if m not in open_set and m not in
closed_set:
                        open_set.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight

                    #for each node m,compare its distance
from start i.e g(m) to the
                    #from start through n node
                    else:
                        if g[m] > g[n] + weight:
                            #update g(m)
                            g[m] = g[n] + weight
                            #change parent of m to n
                            parents[m] = n

                            #if m in closed set,remove and
add to open
                            if m in closed_set:
                                closed_set.remove(m)
                                open_set.add(m)

            if n == None:
                print('Path does not exist!')
                return None

            # if the current node is the stop_node
            # then we begin reconstructin the path from it
to the start_node
            if n == stop_node:
                path = []

                while parents[n] != n:
                    path.append(n)
                    n = parents[n]
```

This lab manual is prepared only for the students understanding and not for commercialization

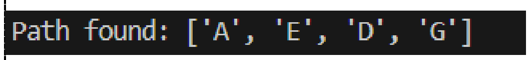
```
                path.append(start_node)

                path.reverse()

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

            # remove n from the open_list, and add it to
closed_list
            # because all of his neighbors were inspected
            open_set.remove(n)
            closed_set.add(n)

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

#define fuction to return neighbor and its distance
#from the passed node
def  get_neighbors(v):
    if v in Graph_nodes:
        return Graph_nodes[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all
nodes
def heuristic(n):
        H_dist = {
            'A': 11,
            'B': 6,
            'C': 99,
            'D': 1,
            'E': 7,
            'G': 0, }

        return H_dist[n]

#Describe your graph here
Graph_nodes = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)],
```

This lab manual is prepared only for the students understanding and not for commercialization

MIT Art Design and Technology University's
MIT School of Computing, Pune
Department of Computer Science and Engineering
BTech Third Year **A.Y.2023-24**

Artificial Intelligence and Machine Learning Lab

	`}`
	`aStarAlgo('A', 'G')`

Outputs	 `Path found: ['A', 'E', 'D', 'G']` (Students can attach extra pages if necessary)

Exercise for Practice

Q1.	How does the A * algorithm work?
Ans:	The A* algorithm works by maintaining open and closed sets of nodes, assigning tentative costs to nodes and calculating heuristics. It selects and explores nodes with the lowest combined cost, moving towards the goal efficiently by considering both actual costs and heuristic estimates.
Q2.	Why is the A* algorithm popular?
Ans	-- -- The A* algorithm is popular due to its guarantee of finding the optimal path, making it widely used in pathfinding and graph traversal. Its versatility allows application in diverse scenarios, ensuring completeness and adaptability. A* is efficient, prioritizing paths using heuristic functions, and its admissibility ensures optimality. The algorithm's well-studied nature, memory efficiency options, and widespread understanding contribute to its enduring popularity in various domains.

This lab manual is prepared only for the students understanding and not for commercialization

Q3	Why A* better than Dijkstra?
Ans	-- A* is particularly advantageous in scenarios where there is a need for efficiency in finding the shortest path. By combining the cost-so-far and the estimated cost-to-go using the heuristic, A* can intelligently guide the search towards the goal while considering the current accumulated cost. This ability to prioritize paths based on a balance of actual cost and estimated remaining cost enables A* to explore fewer nodes than Dijkstra's algorithm in many cases. Moreover, A* excels in scenarios where the search space is large and the goal is distant. The heuristic helps prune unnecessary branches early in the search, leading to faster convergence towards the optimal solution. Dijkstra's algorithm, on the other hand, exhaustively explores all possible paths from the start node, which can become computationally expensive in situations with extensive search spaces. -- --

This lab manual is prepared only for the students understanding and not for commercialization

MIT Art Design and Technology University's
MIT School of Computing, Pune
Department of Computer Science and Engineering
BTech Third Year **A.Y.2023-24**

MIT-ADT UNIVERSITY
PUNE, INDIA
A leap towards World Class Education
(Established by MIT Art, Design and Technology University Act, 2015
(Maharashtra Act No. XXXIX of 2015)

Artificial Intelligence and Machine Learning Lab

Conclusion	In conclusion, the A* algorithm stands out as a powerful and widely utilized solution for pathfinding problems. Its ability to combine an informed heuristic with actual cost information makes it particularly efficient in searching for optimal paths, outperforming algorithms like Dijkstra's in scenarios with large search spaces. A* not only guarantees optimality but also showcases adaptability across various domains, thanks to its versatility and widespread applicability.

Date of Submission:	
Marks out of 10	**Name and Sign of Subject Teacher**
Remark:	

This lab manual is prepared only for the students understanding and not for commercialization

This lab manual is prepared only for the students understanding and not for commercialization