ASSIGNMENT No.

AIM : Implement A* approach for any suitable application.

OBJECTIVE :

- To understand the basic concept of A* approach.
- To implement A* algorithm for shortest path _nding application.

SOFTWARE REQUIREMENTS :

- Linux Operating System
- Python compiler
-

MATHEMATICAL MODEL :

Consider a set S consisting of all the elements related to a program.The mathematical model is given as below,

S=fs,e,X,Y,Fme,DD,NDD,Mem sharedg

Where,

s = Initial State
e = End State
X = Input. Here it is map size,start point,_nish point.
Y = Output.Here output is time required to generate route,actual route and map.
Fme = Algorithm/Function used in program.
for eg.fnextdistance(),estimate(),pathFind()g
DD=Deterministic Data
NDD=Non deterministic Data
Mem shared=Memory shared by processor.

THEORY :

A* Algorithm :

A* is the most widely used approach for path_nding.It is the A* approach that makes use of the function g(n) along with h.Evaluation function f'(n) represents the total cost.

Below is the classic representation of the A* algorithm.

$$f'(n) = g(n) + h'(n) \quad (1)$$

g(n) is the total distance it has taken to get from the starting position to the current location.

h'(n) is the estimated distance from the current position to the goal des-tination/state. A heuristic function is used to create this estimate on how far away it will take to reach the goal state.

f'(n) is the sum of g(n) and h'(n). This is the current estimated shortest path. f(n) is the true shortest path which is not discovered until the A* algorithm is _nished.

In A* approach,the paths are not duplicated, they simply remain as paths that the algorithm hasn't explored yet. A* works by maintaining an open set/open list, it is the collection of nodes the algorithm knows already how to reach (and by what cost), but it hasn't tried expanding them yet.

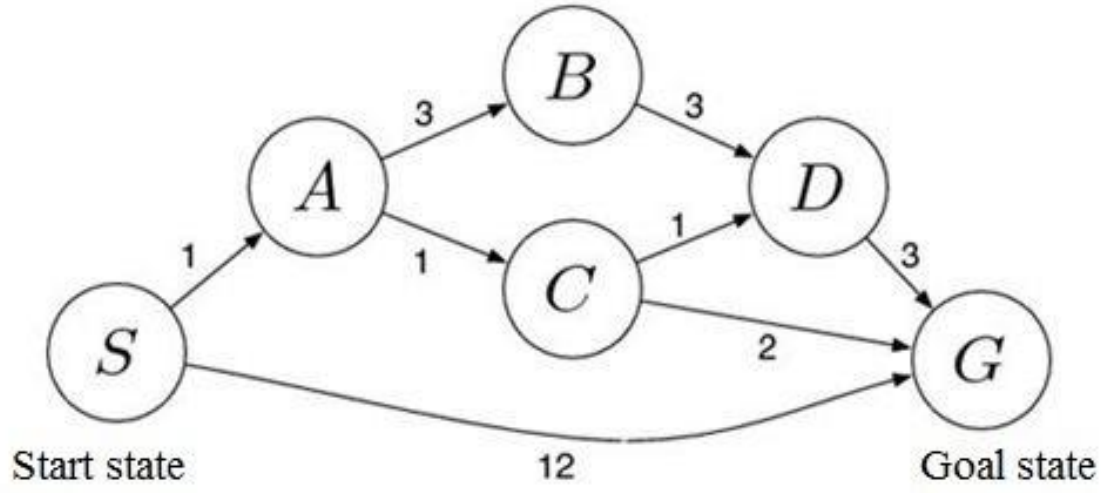
At each iteration the algorithm chooses a node to expand from the open set (the one with the lowest f function - the f function is the sum of the cost the algorithm already knows it takes to get to the node (g) and the the algorithm's estimate of how much it will cost to get from the node to the goal (h, the heuristic).

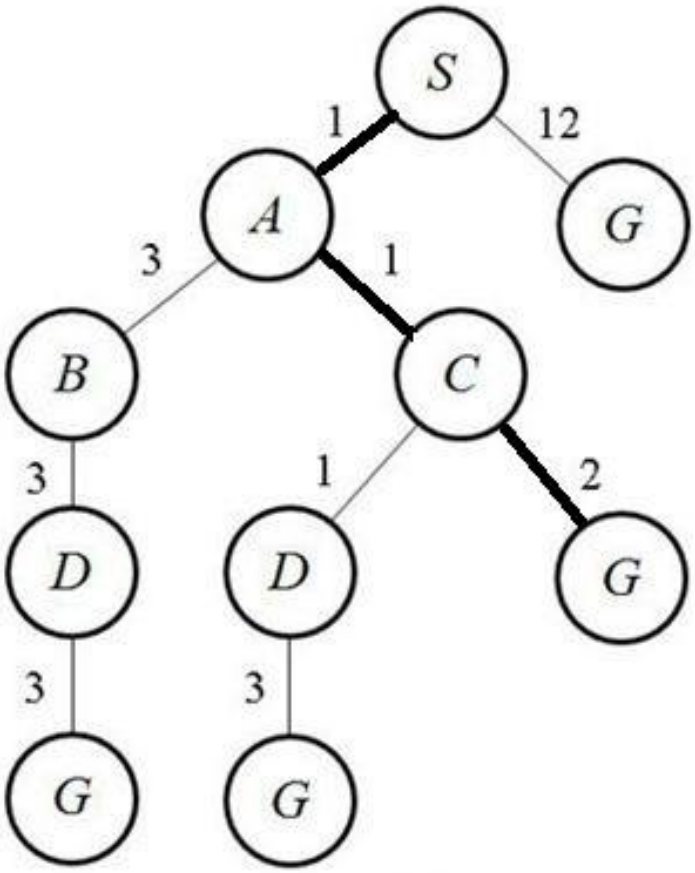
A* Search also makes use of a priority queue which is quite similar to the Uniform Cost Search algorithm:

Insert the root node into the queue.
While the queue is not empty
    Dequeue the element with the highest priority
    (If priorities are same, alphabetically smaller path is chosen)
        If the path is ending in the goal state, print the path and exit
    Else
        Insert all the children of the dequeued element, with f(n) as the priority.

A* Algorithm : Example

Now let us apply the algorithm on the following search tree and see what it gives us. We will go through each iteration and look at the _nal output





State	h1	h2
S	4	3
A	2	2
B	6	5
C	2	1
D	3	2
G	0	0

Fig: A* algorithm Example

Each element of the priority queue is written as [path,f(n)]. We will use h1 as the heuristic, given in the above diagram.

Initialization: f [S , 4] g
Iteration1: f [S->A , 3] , [S->G , 12] g
Iteration2: f [S->A->C , 4] , [S->A->B , 10] , [S->G , 12] g
Iteration3: f [S->A->C->G , 4] , [S->A->C->D , 6] , [S->A->B, 10] , [S->G , 12] g
Iteration4 gives the _nal output as S->A->C->G.

Testing for all functions (For verity input, analyze expected output):

Test Case No.	**Function**	**Input**	**Output**	**Expected Result**
1	**Accept()** Node Elements	Integer (Node Value(g(n)+h(n)) Integer(g'(n))	No error	Node and elements are initialized
2	**Search()**	Node	No error	Searching least distance towards destination
3	**Route()**	Elements	No error	Calculating the route from source to destination
4	**Display()**	Search result and Route result	No error	Route is displayed

Algorithm :

1. Create a search graph G, consisting solely of the start node, no. Putno on a list called OPEN.
2. Create a list called CLOSED that is initially empty.
3. If OPEN is empty, exit with failure.
4. Select the _rst node on OPEN, remove it from OPEN, and put it onCLOSED. Called this node n.
5. If n is a goal node, exit successfully with the solution obtained by tracing a path along the pointers from n to no in G. (The pointers de_ne a search tree and are established in Step 7.)
6. Expand node n, generating the set M, of its successors that are notalready ancestors of n in G. Install these members of M as successors of n in G.
7. Establish a pointer to n from each of those members of M that were not already in G (i.e., not already on either OPEN or CLOSED). Add these members of M to OPEN. For each member, m, of M that was already on OPEN or CLOSED, redirect its pointer to n if the best path to m found so far is through n. For each member of M already on CLOSED,redirect the pointers of each of its descendants in G so that they point
backward along the best paths found so far to these descendants.
8. Reorder the list OPEN in order of increasting f values. (Ties among minimal f values are resolved in favor of the deepest node in the search tree.)
9. Go to Step 3.

CONCLUSION :

Thus, we have implemented A* algorithm for shortest path finding application.

Roll No.	Name	Date Of Performance	Date of Submission	Sign

Program:

```
# A* Shortest Path Algorithm
```

```
# http://en.wikipedia.org/wiki/A*
# FB - 201012256
from heapq import heappush, heappop # for priority queue
import math
import time
import random

class node:
    # current position
    xPos = 0
    yPos = 0
    # total distance already travelled to reach the node
    distance = 0
    # priority = distance + remaining distance estimate
    priority = 0 # smaller: higher priority
    def __init__(self, xPos, yPos, distance, priority):
        self.xPos = xPos
        self.yPos = yPos
        self.distance = distance
        self.priority = priority
    def __lt__(self, other): # for priority queue
        return self.priority < other.priority
    def updatePriority(self, xDest, yDest):
        self.priority = self.distance + self.estimate(xDest, yDest) * 10 # A*
    # give better priority to going straight instead of diagonally
    def nextdistance(self, i): # i: direction
        if i % 2 == 0:
            self.distance += 10
        else:
            self.distance += 14

    # Estimation function for the remaining distance to the goal.
    def estimate(self, xDest, yDest):
        xd = xDest - self.xPos
        yd = yDest - self.yPos
        # Euclidian Distance
        d = math.sqrt(xd * xd + yd * yd)
        # Manhattan distance
        # d = abs(xd) + abs(yd)
        # Chebyshev distance
        # d = max(abs(xd), abs(yd))
        return(d)

# A-star algorithm.
# Path returned will be a string of digits of directions.
def pathFind(the_map, directions, dx, dy, xStart, yStart, xFinish, yFinish):
```

```
closed_nodes_map = [] # map of closed (tried-out) nodes
open_nodes_map = [] # map of open (not-yet-tried) nodes
dir_map = [] # map of directions
row = [0] * n
for i in range(m): # create 2d arrays
    closed_nodes_map.append(list(row))
    open_nodes_map.append(list(row))
    dir_map.append(list(row))

pq = [[], []] # priority queues of open (not-yet-tried) nodes
pqi = 0 # priority queue index
# create the start node and push into list of open nodes
n0 = node(xStart, yStart, 0, 0)
n0.updatePriority(xFinish, yFinish)
heappush(pq[pqi], n0)
open_nodes_map[yStart][xStart] = n0.priority # mark it on the open nodes map

# A* search
while len(pq[pqi]) > 0:
    # get the current node w/ the highest priority
    # from the list of open nodes
    n1 = pq[pqi][0] # top node
    n0 = node(n1.xPos, n1.yPos, n1.distance, n1.priority)
    x = n0.xPos
    y = n0.yPos
    heappop(pq[pqi]) # remove the node from the open list
    open_nodes_map[y][x] = 0
    # mark it on the closed nodes map
    closed_nodes_map[y][x] = 1

    # quit searching when the goal state is reached
    # if n0.estimate(xFinish, yFinish) == 0:
    if x == xFinish and y == yFinish:
        # generate the path from finish to start
        # by following the directions
        path = ''
        while not (x == xStart and y == yStart):
            j = dir_map[y][x]
            c = str((j + directions / 2) % directions)
            path = c + path
            x += dx[j]
            y += dy[j]
        return path

    # generate moves (child nodes) in all possible directions
    for i in range(directions):
```

```
            xdx = x + dx[i]
            ydy = y + dy[i]
            if not (xdx < 0 or xdx > n-1 or ydy < 0 or ydy > m - 1
                    or the_map[ydy][xdx] == 1 or closed_nodes_map[ydy][xdx] == 1):
                # generate a child node
                m0 = node(xdx, ydy, n0.distance, n0.priority)
                m0.nextdistance(i)
                m0.updatePriority(xFinish, yFinish)
                # if it is not in the open list then add into that
                if open_nodes_map[ydy][xdx] == 0:
                    open_nodes_map[ydy][xdx] = m0.priority
                    heappush(pq[pqi], m0)
                    # mark its parent node direction
                    dir_map[ydy][xdx] = (i + directions / 2) % directions
                elif open_nodes_map[ydy][xdx] > m0.priority:
                    # update the priority info
                    open_nodes_map[ydy][xdx] = m0.priority
                    # update the parent direction info
                    dir_map[ydy][xdx] = (i + directions / 2) % directions
                    # replace the node
                    # by emptying one pq to the other one
                    # except the node to be replaced will be ignored
                    # and the new node will be pushed in instead
                    while not (pq[pqi][0].xPos == xdx and pq[pqi][0].yPos == ydy):
                        heappush(pq[1 - pqi], pq[pqi][0])
                        heappop(pq[pqi])
                    heappop(pq[pqi]) # remove the wanted node
                    # empty the larger size pq to the smaller one
                    if len(pq[pqi]) > len(pq[1 - pqi]):
                        pqi = 1 - pqi
                    while len(pq[pqi]) > 0:
                        heappush(pq[1-pqi], pq[pqi][0])
                        heappop(pq[pqi])
                    pqi = 1 - pqi
                    heappush(pq[pqi], m0) # add the better node instead
    return '' # no route found

# MAIN
directions = 8 # number of possible directions to move on the map
if directions == 4:
    dx = [1, 0, -1, 0]
    dy = [0, 1, 0, -1]
elif directions == 8:
    dx = [1, 1, 0, -1, -1, -1, 0, 1]
    dy = [0, 1, 1, 1, 0, -1, -1, -1]
```

```
# map matrix
n = 30 # horizontal size
m = 30 # vertical size
the_map = []
row = [0] * n
for i in range(m):
    the_map.append(list(row))

# fillout the map matrix with a '+' pattern
for x in range(n / 8, n * 7 / 8):
    the_map[m / 2][x] = 1
for y in range(m/8, m * 7 / 8):
    the_map[y][n / 2] = 1

# randomly select start and finish locations from a list
sf = []
sf.append((0, 0, n - 1, m - 1))
sf.append((0, m - 1, n - 1, 0))
sf.append((n / 2 - 1, m / 2 - 1, n / 2 + 1, m / 2 + 1))
sf.append((n / 2 - 1, m / 2 + 1, n / 2 + 1, m / 2 - 1))
sf.append((n / 2 - 1, 0, n / 2 + 1, m - 1))
sf.append((n / 2 + 1, m - 1, n / 2 - 1, 0))
sf.append((0, m / 2 - 1, n - 1, m / 2 + 1))
sf.append((n - 1, m / 2 + 1, 0, m / 2 - 1))
(xA, yA, xB, yB) = random.choice(sf)

print 'Map Size (X,Y): ', n, m
print 'Start: ', xA, yA
print 'Finish: ', xB, yB
t = time.time()
route = pathFind(the_map, directions, dx, dy, xA, yA, xB, yB)
print 'Time to generate the route (s): ', time.time() - t
print 'Route:'
print route

# mark the route on the map
if len(route) > 0:
    x = xA
    y = yA
    the_map[y][x] = 2
    for i in range(len(route)):
        j = int(route[i])
        x += dx[j]
        y += dy[j]
        the_map[y][x] = 3
    the_map[y][x] = 4
```

```
# display the map with the route
print 'Map:'
for y in range(m):
    for x in range(n):
        xy = the_map[y][x]
        if xy == 0:
            print '.', # space
        elif xy == 1:
            print 'O', # obstacle
        elif xy == 2:
            print 'S', # start
        elif xy == 3:
            print 'R', # route
        elif xy == 4:
            print 'F', # finish
    print

raw_input('Press Enter...')
```

Output

Plagiarism Report

Check For Plagiarism

Result:

70% Unique

OBJECTIVE : • To understand the basic concept of A* approach.	- Unique
SOFTWARE REQUIREMENTS : • Linux Operating System • Python	- Unique
of all the elements related to a program.The mathematical	- Unique
Where, s = Initial State e = End State X = Input. Here it	- Unique
is time required to generate route,actual route and map.	- Unique
DD=Deterministic Data NDD=Non deterministic Data Mem shared=Memory	- Unique
widely used approach for path_nding.It is the A* approach	- Unique
function f'(n) represents the total cost. Below is the classic	- Unique

(1) g(n) is the total distance it has taken to get from	- Unique
the estimated distance from the current position to the	- Unique
create this estimate on how far away it will take to reach	- Plagiarized
is the current estimated shortest path. f(n) is the true	- Plagiarized
is _nished. In A* approach,the paths are not duplicated,	- Unique
yet. A* works by maintaining an open set/open list, it is	- Unique
to reach (and by what cost), but it hasn't tried expanding	- Unique
to expand from the open set (the one with the lowest f function	- Unique
knows it takes to get to the node (g) and the the algorithm's	- Unique
the goal (h, the heuristic). A* Search also makes use of	- Unique
Search algorithm: Insert the root node into the queue. While	- Plagiarized
priority (If priorities are same, alphabetically smaller	- Plagiarized
print the path and exit Else Insert all the children of	- Plagiarized
: Example Now let us apply the algorithm on the following	- Unique