18CSC305J- ARTIFICIAL INTELLEGENCE
Experiment-5

DEVELOPING BREDTH FIRST SEARCH AND A* ALGORITHM FOR REAL WORLD PROBLEM

Team Ai 4 life:

Sai Mohit Ambekar (137)

Sadekar Adesh H. (141)

Kapuluru Srinivasulu (142)

Praneet Botke (149)

Ayushi Goenka (151)

Sonia Raja(152)

Aim:

To develop and implement A* algorithm and Best first search for real world problems.

A* Algorithm:

Given a MxN matrix where each element can either be 0 or 1. We need to find the shortest path between a given source cell to a destination cell. The path can only be created out of a cell if its value is 1.

Expected time complexity is O(MN).

Procedure:

1. We start from the source cell and calls BFS procedure.
2. We maintain a queue to store the coordinates of the matrix and initialize it with the source cell.
3. We also maintain a Boolean array visited of same size as our input matrix and initialize all its elements to false.

4. We LOOP till queue is not empty
5. Dequeue front cell from the queue
6. Return if the destination coordinates have reached.
7. For each of its four adjacent cells, if the value is 1 and they are not visited yet, we enqueue it in the queue and also mark them as visited.

Code:

```
import sys

# Check if it is possible to go to (x, y) from the current position. The #
function returns false if the cell is invalid, has value 0 or already visited
def isSafe(mat, visited, x, y):
   return 0 <= x < len(mat) and 0 <= y < len(mat[0]) and \
not (mat[x][y] == 0 or visited[x][y])

# Find the shortest possible route in a matrix `mat` from source cell (i, j) # to
destination cell `dest`.
 # `min_dist` stores the length of the longest path from source to a destination
# found so far, and `dist` maintains the length of the path from a source cell to
# the current cell (i, j).  def findShortestPath(mat, visited, i, j, dest,
min_dist=sys.maxsize, dist=0):     # if the destination is found, update `min_dist`

if (i, j) == dest:
      return min(dist, min_dist)     # set
(i, j) cell as visited     visited[i][j] = 1
# go to the bottom cell
if isSafe(mat, visited, i + 1, j):
      min_dist = findShortestPath(mat, visited, i + 1, j, dest, min_dist, dist +
1)     # go to the right cell     if isSafe(mat, visited, i, j + 1):
      min_dist = findShortestPath(mat, visited, i, j + 1, dest, min_dist, dist +
1)     # go to the top cell     if isSafe(mat, visited, i - 1, j):
      min_dist = findShortestPath(mat, visited, i - 1, j, dest, min_dist, dist + 1)
# go to the left cell     if isSafe(mat, visited, i, j - 1):
```

```
        min_dist = findShortestPath(mat, visited, i, j - 1, dest, min_dist, dist + 1)
# backtrack: remove (i, j) from the visited matrix     visited[i][j] = 0      return
min_dist
 # Wrapper over findShortestPath() function def
findShortestPathLength(mat, src, dest):      # get source cell (i, j)     i, j
= src      # get destination cell (x, y)     x, y = dest      # base case     if not
mat or len(mat) == 0 or mat[i][j] == 0 or mat[x][y] == 0:
        return -1
    # `M × N` matrix
    (M, N) = (len(mat), len(mat[0]))
     # construct an `M × N` matrix to keep track of visited cells
visited = [[False for _ in range(N)] for _ in range(M)]      min_dist =
findShortestPath(mat, visited, i, j, dest)      if min_dist !=
sys.maxsize:
        return min_dist     else:
        return -1
if __name__ =='__main__':
    mat = [ [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 1, 1, 1, 1, 1, 0, 1, 0, 1],
        [0, 0, 1, 0, 1, 1, 1, 0, 0, 1],
        [1, 0, 1, 1, 1, 0, 1, 1, 0, 1],
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 1],
        [1, 0, 1, 1, 1, 0, 0, 1, 1, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 0, 1],
        [0, 1, 1, 1, 1, 1, 1, 1, 0, 0],
        [1, 1, 1, 1, 1, 0, 0, 1, 1, 1],
        [0, 0, 1, 0, 0, 1, 1, 0, 0, 1]
    ]
src = (0, 0)
dest = (7, 5)
min_dist = findShortestPathLength(mat, src, dest)
if min_dist != -1:
        print("The shortest path from source to destination has length", min_dist)
else:
        print("Destination cannot be reached from source")
```

Output:

Best First Search:

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So, both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search.

We use a priority queue to store costs of nodes. So, the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

Algorithm:

1) Create an empty PriorityQueue

   PriorityQueue pq;

2) Insert "start" in pq. pq.insert(start)

3) Until PriorityQueue is empty

u = PriorityQueue.DeleteMin

If u is the goal

Exit

Else

Foreach neighbor v of u

If v "Unvisited"

Mark v "Visited"

pq.insert(v)

Mark u "Examined"

Code:

```
from queue import
PriorityQueue

v = 5
graph = [[] for i in range(v)]
def
best_first_search(source,
target, n):
    visited = [0] * n
    visited[0] = True

pq = PriorityQueue()
pq.put((0, source))

while pq.empty() == False:
    u = pq.get()[1]
    print(u, end=" ")
    if u == target:
        break

for v, c in graph[u]:
    if visited[v] == False:
        visited[v] = True
        pq.put((c, v))
        print()

def addedge(x, y, cost):
    graph[x].append((y,
cost))
    graph[y].append((x,
cost)) addedge(0, 1, 5)
addedge(0, 2, 1)
addedge(2, 3, 2)
addedge(1, 4, 1)
addedge(3, 4, 2)
```

```
    source = 0
    target = 4
    best_first_search(source,
target, v)
```

Output:

Result:

Both Best First Search and A* algorithm for real world problems were successfully developed and implemented using Python 3.