Vellore Institute of Technology
(Deemed to be University under section 3 of UGC Act, 1956)

ARTIFICIAL INTELLIGENCE

DIGITAL ASSIGNMENT

SLOT:B2+TB2

COURSE CODE:BCSE306L

DONE BY:

SHIVANK PANDEY:21BKT0021

SHIVAM AGARWAL:21BKT0190

QUESTION-1

OVERVIEW:

MAZE PROBLEM:

The 8x8 maze problem involves navigating through an 8x8 grid with obstacles from a starting point to a goal point. The objective is to find a path from the starting point to the goal point while avoiding obstacles. Typically, this problem is tackled using algorithms like depth-first search, breadth-first search, or A* search to efficiently explore the maze and find the optimal or near-optimal path. The challenge lies in efficiently exploring the maze space to find a solution while minimizing the number of steps taken and avoiding dead ends.

A* SEARCH ALGORITHM:

A* search is a pathfinding algorithm that efficiently finds the shortest path from a starting point to a goal point in a graph or grid. It uses a combination of actual cost and a heuristic estimate to guide its search. By prioritizing nodes closer to the goal, it efficiently explores the search space, making it suitable for tasks like maze solving and route planning.

Here's how it works:

Initialize an open list of nodes to explore, starting with the initial node.

While the open list is not empty:

Choose the node with the lowest total cost (combination of actual cost from the start node and heuristic estimate to the goal) from the open list.

If this node is the goal, the search is complete, and the path is found.

Otherwise, expand the chosen node by considering its neighboring nodes.

For each neighbor, calculate its total cost and add it to the open list if it hasn't been visited or has a lower cost than before.

Repeat until either the goal is found or the open list is empty.

A* guarantees the shortest path if certain conditions are met: the heuristic function is admissible (never overestimates the true cost to reach the goal) and consistent (satisfies the triangle inequality). It efficiently explores the search space by prioritizing nodes that are closer to the goal, making it suitable for solving problems like pathfinding in mazes or maps.

METHODS:

PSEUDO CODE:

```
function AStar(maze, start, goal):
    open_list = priority queue containing start node
    closed_list = empty set
    while open_list is not empty:
        current_node = node from open_list with lowest f_cost
        if current_node is goal:
            return reconstruct_path(current_node)
        remove current_node from open_list
        add current_node to closed_list
        for each neighbor of current_node:
            if neighbor is not traversable or neighbor is in closed_list:
                continue
            tentative_g_cost = current_node.g_cost + distance between current_node and neighbor
            if neighbor is not in open_list or tentative_g_cost < neighbor.g_cost:
                neighbor.parent = current_node
                neighbor.g_cost = tentative_g_cost
                neighbor.h_cost = heuristic(neighbor, goal)
                neighbor.f_cost = neighbor.g_cost + neighbor.h_cost
            if neighbor is not in open_list:
                add neighbor to open_list
    return failure (no path found)
function reconstruct_path(current_node):
    path = empty list
    while current_node is not null:
        add current_node to path
        current_node = current_node.parent
    return reversed(path)
```

NOTE:

f_cost is the total cost of a node (actual cost from start + heuristic estimate to goal).

g_cost is the actual cost from the start node to the current node.

h_cost is the heuristic estimate from the current node to the goal.

heuristic(node, goal) is a function that estimates the cost from the current node to the goal.

The neighbor refers to adjacent cells of the current node in the maze.

The algorithm maintains a priority queue (open list) to efficiently select the next node with the lowest total cost.

FLOWCHART:

[Start] -> [Initialize open list with start node] -> [Initialize closed list as empty]

While [open list is not empty]:

-> [Choose node with lowest f_cost from open list]

```
-> If [current node is goal],
-> [Reconstruct path]
-> [End]
-> [Remove current node from open list]
-> [Add current node to closed list]
-> For each [neighbor of current node]:
-> If [neighbor is not traversable or in closed list],
-> [Continue loop]
-> [Calculate tentative g_cost]
-> If [neighbor is not in open list or tentative_g_cost <
neighbor.g_cost],
-> [Update neighbor's information]
-> If [neighbor is not in open list],
-> [Add neighbor to open list]
[End]
```

IMPLEMENTATION:

CODE:

```
public class MAZE {
        final int N = 8;
    void printSolution(int sol[][])
```

```
        {
        System.out.println("SOLUTION EXISTS");
        System.out.println();
        String[][] mazerep=new String[8][8];
                // for (int i = 0; i < N; i++) {
                //      for (int j = 0; j < N; j++)
                //              System.out.print(" " + sol[i][j] + " ");
                //      System.out.println();
                // }
        for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++)
                                if(sol[i][j]==1){
                    mazerep[i][j]="PATH";
                }
                else{
                    mazerep[i][j]="--";
                }
                 }
        for (int i = 0; i < N; i++) {
                        for (int j = 0; j < N; j++)
                                System.out.print(" " +mazerep[i][j]+ " ");
                        System.out.println();
                 }
        }
        boolean isSafe(int maze[][], int x, int y)
        {
                return (x >= 0 && x < N && y >= 0 && y < N &&
maze[x][y] == 1);
```

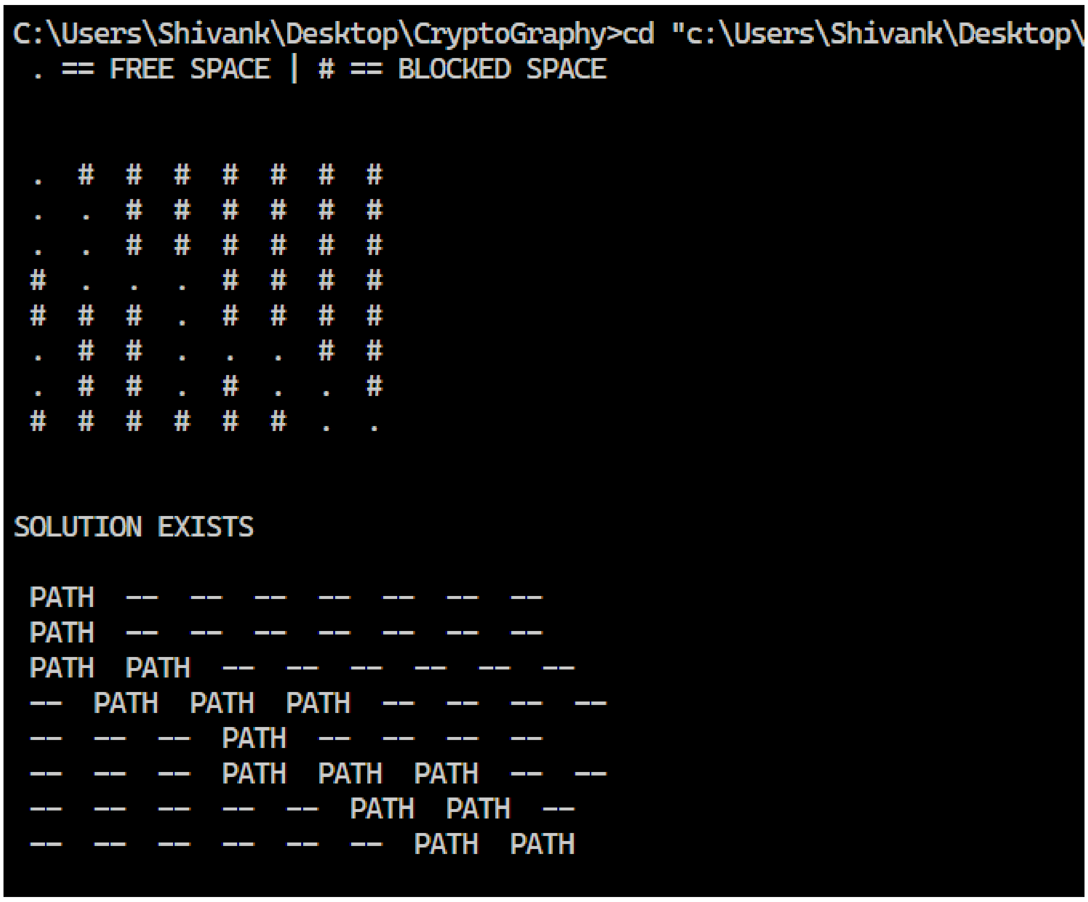
```
}
boolean solveMaze(int maze[][])
{
        int sol[][] = {
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
   { 0,0,0,0,0,0,0,0},
};

        if (solveMazeUtil(maze, 0, 0, sol) == false) {
                System.out.print("Solution doesn't exist");
                return false;
        }
        printSolution(sol);
        return true;
}
boolean solveMazeUtil(int maze[][], int x, int y,
                                        int sol[][])
{
        if (x == N - 1 && y == N - 1) {
                sol[x][y] = 1;
                return true;
        }
```

```
            if (isSafe(maze, x, y) == true) {
                    sol[x][y] = 1;
                    if (solveMazeUtil(maze, x + 1, y, sol))
                            return true;
                    if (solveMazeUtil(maze, x, y + 1, sol))
                            return true;
                    sol[x][y] = 0;
                    return false;
            }
            return false;
    }
    public static void main(String args[])
    {
            MAZE rat = new MAZE();
            int maze[][] = {
    { 1,0,0,0,0,0,0,0},
    { 1,1,0,0,0,0,0,0},
    { 1,1,0,0,0,0,0,0},
    { 0,1,1,1,0,0,0,0},
    { 0,0,0,1,0,0,0,0},
    { 1,0,0,1,1,1,0,0},
    { 1,0,0,1,0,1,1,0},
    { 0,0,0,0,0,0,1,1},
};
System.out.println(" . == FREE SPACE | # == BLOCKED SPACE");
System.out.println();
System.out.println();
String[][] mazerep=new String[8][8];
```

```
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            if(maze[i][j]==1){
                mazerep[i][j]=".";
            }
            else{
                mazerep[i][j]="#";
            }
        }
    }
    for(int i=0;i<8;i++){
        for(int j=0;j<8;j++){
            System.out.print(" "+mazerep[i][j]+" ");
        }
        System.out.println();
    }
    System.out.println();
    System.out.println();
                rat.solveMaze(maze);
         }
}
```

OUTPUT:

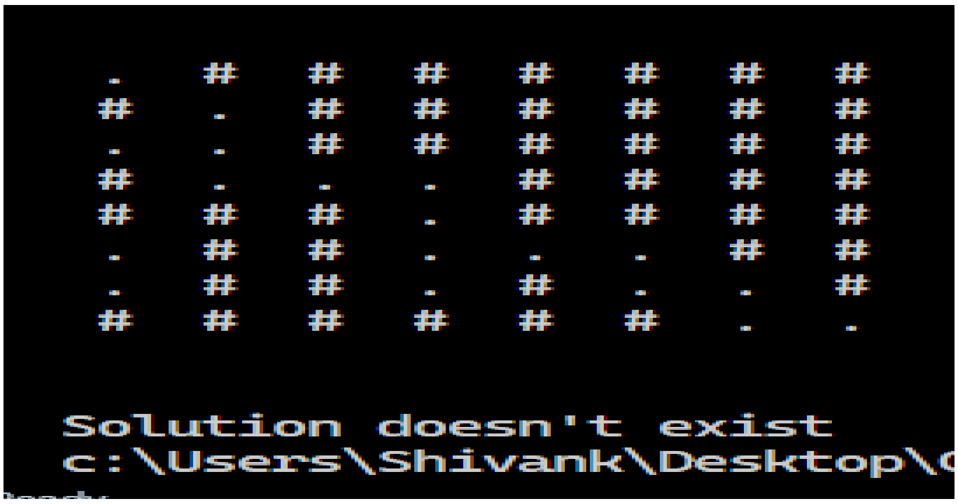


```
C:\Users\Shivank\Desktop\CryptoGraphy>cd "c:\Users\Shivank\Desktop\
 . == FREE SPACE | # == BLOCKED SPACE


 .  #  #  #  #  #  #  #
 .  .  #  #  #  #  #  #
 .  .  #  #  #  #  #  #
 #  .  .  .  #  #  #  #
 #  #  #  .  #  #  #  #
 .  #  #  .  .  .  #  #
 .  #  #  .  #  .  .  #
 #  #  #  #  #  #  .  .


SOLUTION EXISTS

 PATH  --  --  --  --  --  --  --
 PATH  --  --  --  --  --  --  --
 PATH  PATH  --  --  --  --  --  --
 --  PATH  PATH  PATH  --  --  --  --
 --  --  --  PATH  --  --  --  --
 --  --  --  PATH  PATH  PATH  --  --
 --  --  --  --  --  PATH  PATH  --
 --  --  --  --  --  --  PATH  PATH
```



```
    .   #   #   #   #   #   #   #
    #   .   #   #   #   #   #   #
    .   .   #   #   #   #   #   #
    #   .   .   .   #   #   #   #
    #   #   #   .   #   #   #   #
    .   #   #   .   .   .   #   #
    .   #   #   .   #   .   .   #
    #   #   #   #   #   #   .   .


  Solution doesn't exist
  c:\Users\Shivank\Desktop\C
```

REFERENCES:

A Search Algorithm*:

Online Courses: Look for courses on platforms like Coursera, Udemy, or edX that specifically cover algorithms and data structures. Courses such as "Algorithmic Toolbox" on Coursera or "Algorithms, Part I" on Princeton University's Coursera specialization delve into A* and other search algorithms.

Research Papers: Academic papers provide rigorous explanations of A* and its variations. For example, "Anytime A* for Large Scale Path Planning" by Koenig and Likhachev offers insights into optimizing A* for real-world applications.

Books: Textbooks like "Artificial Intelligence: A Modern Approach" by Russell and Norvig discuss A* comprehensively, covering its theoretical foundations and practical implementations.

Java Maze Solving:

Community Forums: Engage with Java programming communities such as Reddit's r/javahelp or Stack Overflow's Java tag. These platforms allow you to ask questions, seek advice, and learn from experienced developers.

Coding Challenges: Platforms like LeetCode, CodeSignal, or HackerRank host coding challenges that include maze-solving problems. Participating in these challenges not only sharpens your problem-solving skills but also exposes you to various maze-solving approaches.

Online Guides and Tutorials: Explore online tutorials and guides on maze-solving algorithms in Java. Websites like GeeksforGeeks, Baeldung, and Tutorialspoint offer step-by-step explanations, code examples, and practical tips for implementing maze-solving algorithms.

By leveraging these detailed resources, you can deepen your understanding of A* and maze-solving algorithms in Java, gaining both theoretical knowledge and practical skills.