A logo with a blue circle and red stripes

Description automatically generated

**T.C.**

**MARMARA UNIVERSITY**

**FACULTY of ENGINEERING**

**COMPUTER ENGINEERING DEPARTMENT**

CSE4082 Artificial Intelligence Report

Title of the Project

*“Maze Solver 5000“*

Group Members

Mert Özincegedik – 150119643  
Aimen Daddi -150119879

1. Introduction

This is a maze solver software application that will take information about the maze to create the walls, traps and the goal states. The program will start solving the maze with different algorithms and put the results in. And then the program will try to find best ways to get the most points by avoiding traps and finding the least cost solution. For this, we will use graph search algorithms that are taught in the course.

1. Problem Description

**2.1 Creating the maze**

We will put the necessary information to generate the maze in a text file. The structure of the text file can be anything, for example when it reads the file and sees “size”, it will expect two numbers under that which are rows and columns. Such as 2 3 which means the size of the maze is 2x3

For the start location, when the program reads the line “start” it will expect two numbers such as 4 5 which means the start is at row 4 and column 5.

When it reads “traps” it will expect two numbers side to side like we did on start. But here we can put multiple traps on each line. For example:

2 4  
3 6  
5 2  
  
That means the traps are at row 2, column 4 and row 3 column 6 and lastly row 5 column 2  
We can put however much traps we want as long as they are not outside of the reach (for example if the size is 3\*3 but we try to put trap on 4x2 it will give error)

Then we look at “walls” under that the program will expect “row x” and then “column y” and under those it will expect a number for example:  
walls  
row 1  
2 /// it will put wall on A12 right side of the square  
row 2  
1 4 5 /// it will put wall on A21, A24 and A25, right side of the squares  
column 3  
2 6 /// it will put wall on A23 and A63 down side of the squares  
column 4  
1 3 6 /// it will put wall on A14, A34, and A64 down side of the squares  
  
In here it will put the wall on row 1 and column 2 and it will always put it to the right side of the square for the rows and down side of the square for columns.  
  
Lastly we have “goals” part, it is basically the same idea as “traps” it will be put in so that there are 2 numbers on each line after reading goals such as 4 5 which means A45 is a goal state  
  
After providing these as “maze.txt” file, the program will start to generate the maze. You can look at our maze.txt file to understand the structure completely. It is in the project directory itself.

**2.2 Solving the maze**We have implemented six search algorithms to solve the given maze. Each algorithm explores the maze differently, aiming to find the optimal path from the start to the goal states. **a. Depth First Search (DFS):**

* DFS explores as far as possible along each branch before backtracking.
* The algorithm uses a stack to keep track of nodes to visit.
* The solution path is printed to the command line.

**b. Breadth First Search (BFS):**

* BFS explores all the neighbor nodes at the present depth prior to moving on to nodes at the next depth level.
* The algorithm uses a queue to keep track of nodes to visit.
* The solution path is printed to the command line.

**c. Iterative Deepening (IDS):**

* IDS performs depth-limited searches with increasing depth limits until a solution is found.
* The algorithm uses a depth-first search with a depth limit.
* The solution path is printed to the command line.

**d. Uniform Cost Search (UCS):**

* UCS explores the node with the lowest cost, considering the accumulated cost from the start node.
* The algorithm uses a priority queue based on the cost function.
* The solution path is printed to the command line.

**e. Greedy Best First Search (GBFS):**

* GBFS selects the node that is closest to the goal based on a heuristic function.
* The algorithm uses a priority queue based on the heuristic function.
* The solution path is printed to the command line.

**f. A\* Heuristic Search (A\*):**

* A\* combines the benefits of UCS and GBFS, using both the cost function and the heuristic function.
* The algorithm uses a priority queue based on the sum of cost and heuristic.
* The solution path is printed to the command line

1. Classes and Method Structure

We have designed our software in a way that it will have Maze class, Graph class and Node class

**3.1 Maze Class 3.2 Node Class  
3.1.1 Attributes and Initialization 3.2.1 Attributes and Initialization:**size = [] x = 0  
verticalWalls = [[]] y = 0  
horizantalWalls = [[]] cost = 0  
traps = [[]] parent = None  
start = [] east = None  
goals = [] south = None  
**3.1.2 Methods:** west = None  
read\_maze() //reads maze from text north = None  
Size(x, y)//sets maze size heuristic = 0  
Walls(walls)// sets maze walls **3.2.2 Methods:**  
Traps(traps) // sets maze traps check\_equality(x, y) // returns if x, y are same  
Start(start) // sets maze start \_\_str\_\_() //to convert to string and return it  
Goals(goals) // sets maze goals   
Blocked(row, column direction) // sets the outer walls of the maze  
  
**3.3 Graph Class  
3.3.1 Attributes and Initialization**nodes = []  
maze = None // in initialization we set this equal to mazeOutput  
root = createNode(self.maze.start[0], self.maze.start[1])  
maximum\_depth = findMaxDepth() – 1  
root.cost = 0  
**3.3.2 Methods**createNode(x, y) // Here we create node with x and y coordinates  
ifNodeExists(x,y) // Checks if a node with given coordinates already exists  
findMaxDepth() // Finds the maximum depth in the graph (value of m)  
nodeCost(x, y) // Get the cost of a node with given coordinates  
clearParents() // Clears the parents references for all nodes  
putHeuristic() // It will calculate the heuristic value and put it in the node.heuristic  
  
These are our class structures, we also have methods for each of the algorithms for solving the mazes, but they are not in a class and they are just functions.  
Here are the name of those:  
  
dfs() // Depth-First Search algorithm  
bfs() // Breadth-First Search algorithm  
ids() // Iterative-Deepening Search algorithm  
ucs() // Uniform-Cost Search algorithm  
gbfs() // Greedy-Best First Search algorithm  
astar() // A\* algorithm

1. Output Result

A black screen with many small colored lines

Description automatically generated

Thank you for reading, I hope you like it.

Regards,  
Mert Özincegedik, Aimen Daddi