T.C.

MARMARA UNIVERSITY

FACULTY of ENGINEERING

COMPUTER ENGINEERING DEPARTMENT

CSE4082 - Homework 1

GROUP MEMBERS

150115021 - Öznur AKYÜZ

150115069 - Muhammet ŞERAMET

11 December 2019

CONTENTS

1. **Input File Format** 3
2. **Object Classes** 3
   - 2.1. **MazeReader** 3
   - 2.2. **Node** 3
   - 2.3. **Square** 3
     - 2.3.1. **Variables** 3
     - 2.3.2. **Methods** 4
   - 2.4. **Maze** 4
     - 2.4.1. **Variables** 4
     - 2.4.2. **Methods** 4
   - 2.5. **Wall** 4
   - 2.6. **Utils** 5
     - 2.6.1. **Methods** 5
   - 2.7. **SearchAlgorithms** 7
     - 2.7.1. **Methods** 7
   - 2.8. **Comparators** 8
   - 2.9. **Main** 8
3. **Outputs** 8
   - 3.1. **Start Screen** 8
   - 3.2. **Depth-First Search** 9
   - 3.3. **Breadth-First Search** 10
   - 3.4. **Iterative Deepening** 11
   - 3.5. **Uniform Cost Search** 12
   - 3.6. **Greedy Best First Search** 13
   - 3.7. **A* Heuristic Search** 14
4. **Execution with a Different Maze** 15
   - 4.1. **Depth-First Search** 15
   - 4.2. **Breadth-First Search** 16
   - 4.3. **Iterative Deepening** 16
   - 4.4. **Uniform Cost Search** 17
   - 4.5. **Greedy Best First Search** 17
   - 4.6. **A* Heuristic Search** 18
5. **Comparison of Algorithms and Conclusion** 19

1. Input File Format

The first line of the input file consists of 2 numbers, that are mapped to row and column indexes, representing the dimensions of the maze. The remaining lines consist of 1 letter and 2 numbers. The letter indicates the expression in the line(the type of wall or square), while the numbers indicate the row and column indexes, respectively. Meanings of the letters are;

v: vertical walls in the maze

h: horizontal walls in the maze

s: the starting square of the maze

t: trap squares in the maze

g: goal squares in the maze

2. Object Classes

2.1. MazeReader

As a beginning, we read the first line only. This line represents the row and column dimensions of the maze frame, respectively. After this line, lines consist of 1 letter and 2 numbers. The letter indicates the type of the line, while the numbers indicate the row and column indexes, respectively. For example, when we read the “v 3 4”; we create a node that has row number with 3 and column number with 4. After creating the Node object, it is added into “VerticleWalls ArrayList”. All the rules that are applied at the “VerticleWalls” part are the same for “HorizontalWalls”, “trapSquares” and “goalSquares”. For start, which expressed with “s”, we created the node object and this object is equal to startSquare.

2.2. Node

The Node class is an object class that we create using the rows and columns in the maze file that we read in MazeReader and that will be used for squares.

2.3. Square

2.3.1. Variables

Square class refers to each square on the maze and extends JPanel for GUI. There are 4 boolean variables: up, down, left, right for the walls that can be placed around each square. Other boolean variables can be listed as: 'added' to indicate that they are added to the frontier, 'expanded' to indicate that the square is expanded in GUI, 'goal' to indicate that the goal square is reached in GUI, 'path' to indicate that the square in the algorithm' solution path in GUI. Row

and column variables indicate the location of the square in the maze, and depth expresses the depth of the square in the Depth-First Search and Iterative Deepening algorithms. The SquareType string indicates the type of square, goal square is defined as 'g', start square as 's', trap square as 't', none of them as 'n'. walls LinkedList is defined for lines that will be drawn to the panel.

2.3.2. Methods

void addWall(int x1, int y1, int x2, int y2, Color color): Create a Wall object and add to the linked list.

void check(): Based on panels' up, down, right and left attribute, the addWall method is called. If one of them is true, addWall will be called with BLACK color, else LIGHT_GRAY color.

void paintComponent(Graphics g): In order to show all panels on the frame, some variables are controlled. For example, if the panel is goal state, it's color will be red, if the panel's square variable is equal to 't, our panel has a letter with "T".

2.4. Maze

2.4.1. Variables

Maze class refers to the maze itself and extends JFrame for GUI. There are 4 Node ArrayList for walls, goal and trap squares. startSquare initializes the square we create from startNode. Row and column are used to adjust the frame according to the dimensions of the maze. Finally, the squares array will hold all the square objects created.

2.4.2. Methods

void createMazeSquares(): After the frame creation process, we filled the squares array using the mazeReader object. We changed the squareType for start, goal and trap squares. For verticleWalls, we set the "right" field of the current square and the "left" field of the square to the right to "true". For horizontalWalls , we set the "down" field of the current square and the "up" field of the square to the right to "true". Finally, we added the outer walls of the maze.

2.5. Wall

Wall is an object class that has color and 4 different coordinates fields, and represent walls in the maze. Each wall is drawn based on the coordinates from x1 to x2 and from y1 to y2, with color attribute.

2.6. Utils

The Utils class includes a number of static methods that we will use for algorithms.

2.6.1. Methods

static void checkWestSquare(Maze maze, Square frontierSquare, ArrayList<Square> frontierSquareList, Object frontier, int algo_chosen): In this method, we check the square in the west of the square we pull from the frontier. If there is no wall to the left of the frontierSquare, we add the westSquare to expandedFrontierSquareList to create the new path. With the algo_chosen parameter, we keep information about which search algorithm the method is called to. If algo_chosen is equal to 1, this means that Depth-First Search is used. If it is equal to 2, Breadth-First Search, Uniform Cost Search, Greedy Best First Search, or A* Star Search is used, if it is equal to 3, the Iterative Deepening algorithm is used.

If algo_chosen is equal to 1 and westSquare is not expanded, we push expandedFrontierSquareList to the frontier and set the "added" field to true, indicating that we added westSquare to the frontier and set the depth of westSquare by increasing the depth of frontierSquare by 1.

If algo_chosen is equal to 2 and westSquare is not added, we push expandedFrontierSquareList to the frontier and set the "added" field to true, indicating that we added westSquare to the frontier.

If algo_chosen is equal to 3, we check if westSquare is included in the frontierSquareList. If westSquare is not included in the frontierSquareList, we push expandedFrontierSquareList to the frontier, set the depth of westSquare by increasing the depth of frontierSquare by 1 and set the "added" field to true, indicating that we added westSquare to the frontier.

static void checkSouthSquare(Maze maze, Square frontierSquare, ArrayList<Square> frontierSquareList, Object frontier, int algo_chosen): All transactions are on the checkWestSquare will take place at checkSouthSquare. “Left” keywords in checkWestSquare will be “Down” here. maze.getSquares()[frontierSquare.getRow()][frontierSquare.getColumn() - 1] will be maze.getSquares()[frontierSquare.getRow() + 1][frontierSquare.getColumn()].

static void checkEastSquare(Maze maze, Square frontierSquare, ArrayList<Square> frontierSquareList, Object frontier, int algo_chosen): All transactions are on the checkWestSquare will take place at checkEastSquare. “Left” keywords in checkWestSquare

will be “Right” here. maze.getSquares()[frontierSquare.getRow()][frontierSquare.getColumn() - 1] will be maze.getSquares()[frontierSquare.getRow()][frontierSquare.getColumn() + 1].

static void checkNorthSquare(Maze maze, Square frontierSquare, ArrayList<Square> frontierSquareList, Object frontier, int algo_chosen): All transactions are on the checkWestSquare will take place at checkNorthSquare. “Left” keywords in checkWestSquare will be “Up” here. maze.getSquares()[frontierSquare.getRow()][frontierSquare.getColumn() - 1] will be maze.getSquares()[frontierSquare.getRow() - 1][frontierSquare.getColumn()].

static void printResult(Maze maze, String algorithmName, ArrayList<Square> frontierSquareList, ArrayList<Square> expandedSquareList): This method writes solution path, expanded squares and path cost to the output file.

private static int calculatePathCost(ArrayList<Square> path): In this method, we calculate the cost of the solution path. Since the start square is located on a path, we initialize the cost variable to -1. If it is a square trap, we increase the cost by 7, if not, we increase the cost by 1.

private static void addFrontier(Object frontier, ArrayList<Square> squareArrayList): In this method, if one of the check*Direction*Square methods has found a non-searched square, we add its path to the frontier.

private static ArrayList<Square> removeFrontier(Object frontier): In this method, we pull the path element from the frontier.

static void addStartSquare(Maze maze, Object frontier): In this method, we add start square to the frontier to start searching.

static void checkSquares(Maze maze, Square frontierSquare, ArrayList<Square> path, Object frontier, int algo_chosen): The method calls all check*Direction*Square methods to check the squares around the frontierSquare. In this project, our direction order is East, South, West, North.

static void search(String algorithmName, Queue<ArrayList<Square>> frontier, Maze maze, ArrayList<Square> expandedSquareList): In this method, we first add the start square to the frontier. If Frontier is empty before finding any goal, it means that there is no solution. So while the frontier is not empty while loop will execute. The square is pulled from the frontier in the loop, the path is Square ArrayList because it holds expanded squares on the path until it

reaches the square. It checks the squares around the frontierSquare by calling the checkSquares method and adds new squares that can be pushed to the frontier. Additionally, since this square is being searched, it will change the expanded field to true and add it to expandedSquareList. Finally, if the frontierSquare is a goal square, the loop will terminate and the results will be written to the file by calling the printResult method.

static int costFunc(ArrayList<Square> path): In this method, we calculate the path length from the root to the destination node.

static int heuristicFunc(Maze maze, ArrayList<Square> path): In this method, we calculate the cost which is the heuristic prediction of the cost from node to the goal. We calculate the cost for all goal nodes, but we return the minimum cost.

2.7. SearchAlgorithms

In this class, we implemented the search algorithms that we will use in a maze.

2.7.1. Methods

static void BFS(Maze maze): We created the expandedSquareList ArrayList variable that holds expanded squares and the Queue variable which is used as the frontier in the Breadth-First Search algorithm, and called the search() method in the Util class with the maze object and other variables.

static boolean DFS(int depthLimit, Maze maze): In this method, we start by initializing the default values so that they can be used if called for DFS. Then we push the start square to the frontier, which is a Stack object. Unless the frontier is empty, we will search. If the algorithm is called with Iterative Deepening and not directly with DFS, the depthLimit value will be different from Integer.MAX_VALUE. In this case, we need to set the depths of the squares on the path. In addition, default values should be replaced with Iterative Deepening values. For all depths smaller than DepthLimit, we must call the search () method. If we have reached a goal state, we must end the search by printing the results to the file. If DFS has scored a goal for Iterative Deepening, set IDControl to true and return the ID method to terminate the loop.

static void ID(Maze maze): We have defined the IDControl variable to call DFS for each depth. If this variable is false, the DFS method will be called with depthLimit, if the solution is not found, the fields will be set to false and DFS will be called again. If this variable is true, the loop will be broken, and the result will be found.

static void uniformCost(Maze maze): We created the expandedSquareList ArrayList variable that holds expanded squares and the PriorityQueue variable with UniformCostComparator class object which is used as the frontier in the Breadth-First Search algorithm, and called the search() method in the Util class with the maze object and other variables.

static void greedyBestFirst(Maze maze): We created the expandedSquareList ArrayList variable that holds expanded squares and the PriorityQueue variable with GreedyComparator class object which is used as the frontier in the Breadth-First Search algorithm, and called the search() method in the Util class with the maze object and other variables.

static void aStar(Maze maze): We created the expandedSquareList ArrayList variable that holds expanded squares and the PriorityQueue variable with AStarComparator class object which is used as the frontier in the Breadth-First Search algorithm, and called the search() method in the Util class with the maze object and other variables.

2.8. Comparators

Comparators consists of 3 different classes: UniformCostComparator, GreedyComparator, AStarComparator. These are the Comparator types that implement the Comparator interface and will be used for PriorityQueue.

2.9. Main

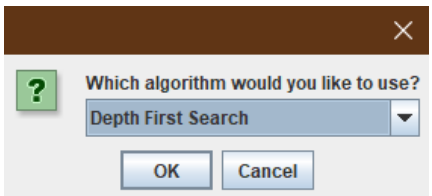
We created an object from our MazeReader class and gave our maze.txt file as a parameter to the readMazeFile () method, so we received our inputs to create a maze. After reading Maze, we have received one of the algorithm selections as user input with using JOptionPane. We have designed the main method as retrieving user input after running the algorithm until the user presses the "Cancel" button.

3. Outputs

3.1. Start Screen

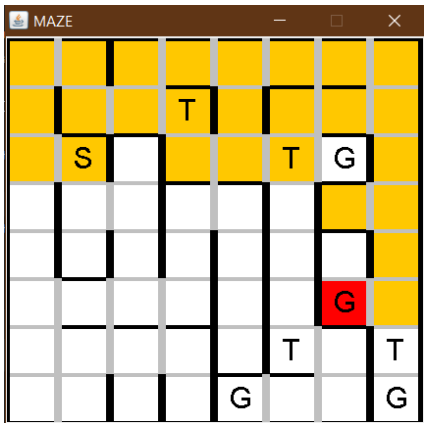
When we execute the main method of the program, the start window will appear.

There exist 2 options: We can exit from the program with the “Cancel” option or execute it after selecting one of the algorithms with the “OK” option.

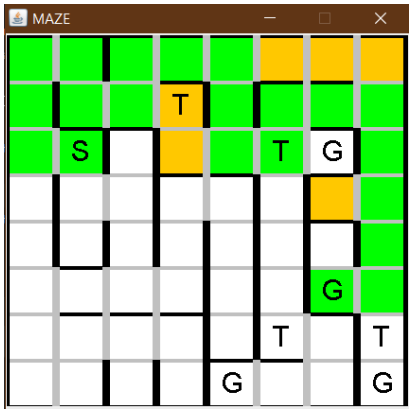


3.2. Depth-First Search

When we choose Depth First Search, the maze window will appear as follows:



Firstly, expanded squares will be orange color constantly. When we found the goal state, the goal state will be red color. If the algorithm finds the solution, squares in the solution path will be painted in green.

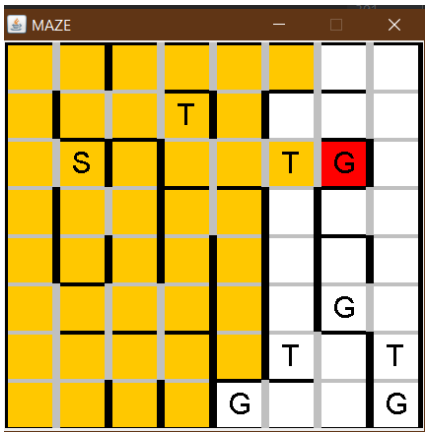


After this process, the start window will appear again. The result of the algorithm is shown in the output.txt as follows:

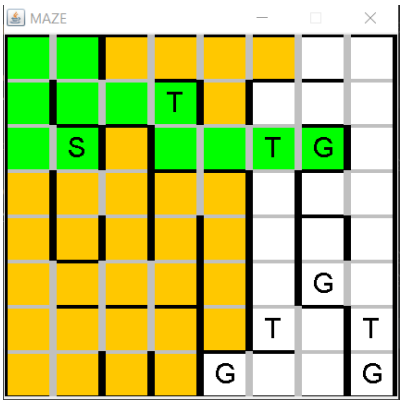
```
================================================================================
Date 04/12/2019 01:44:57
Depth First Search

Solution Path :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (0, 2) => (0, 3) =>
(0, 4) => (1, 4) => (2, 4) => (2, 5) => (1, 5) => (1, 6) => (1, 7) => (2, 7) => (3, 7) =>
(4, 7) => (5, 7) => (5, 6)

The List of Expanded Nodes :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (0, 2) => (0, 3) =>
(0, 4) => (1, 4) => (2, 4) => (2, 3) => (1, 3) => (2, 5) => (1, 5) => (1, 6) => (1, 7) =>
(0, 7) => (0, 6) => (0, 5) => (2, 7) => (3, 7) => (3, 6) => (4, 7) => (5, 7) => (5, 6)

The cost : 26

================================================================================
```

3.3. Breadth-First Search

When we choose Breadth-First Search, the maze window will appear as follows:



The way the solution is searched will proceed as in Depth First Search. If the algorithm finds the solution, squares in the solution path will be painted in green.

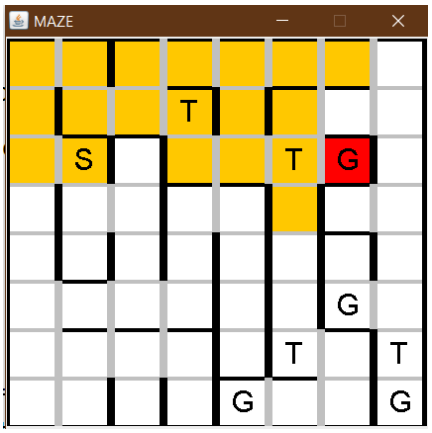


After this process, the start window will appear again. The result of the algorithm is shown in the output.txt as follows:

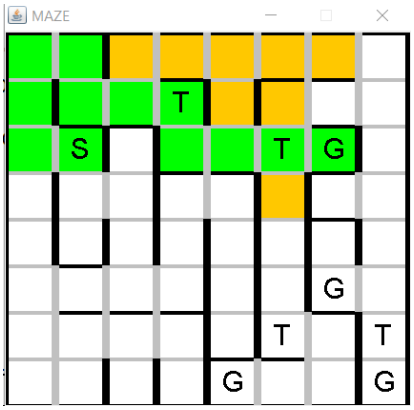
```
================================================================================================
Date 04/12/2019 01:45:24
Breadth First Search

Solution Path :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (1, 3) => (2, 3) =>
(2, 4) => (2, 5) => (2, 6)

The List of Expanded Nodes :
(2, 1) => (3, 1) => (2, 0) => (2, 0) => (3, 2) => (4, 1) => (3, 0) => (1, 0) => (4, 2) =>
(2, 2) => (4, 0) => (0, 0) => (5, 2) => (5, 0) => (0, 1) => (5, 3) => (5, 1) => (5, 1) =>
(6, 0) => (1, 1) => (4, 3) => (5, 0) => (5, 0) => (6, 1) => (7, 0) => (1, 2) => (3, 3) =>
(6, 2) => (7, 1) => (1, 3) => (0, 2) => (3, 4) => (6, 3) => (7, 2) => (7, 0) => (2, 3) =>
(0, 3) => (4, 4) => (7, 3) => (2, 4) => (0, 4) => (5, 4) => (2, 5) => (1, 4) => (0, 5) =>
(6, 4) => (2, 6)

The cost : 23

================================================================================================
```

3.4. Iterative Deepening

When we choose Iterative Deepening, the maze window will appear as follows:



The way the solution is searched will proceed as in Depth First Search. The only difference is that the Iterative Deepening repaints the maze frame for every increasing depth until the solution path is found. If the algorithm finds the solution, squares in the solution path will be painted in green.

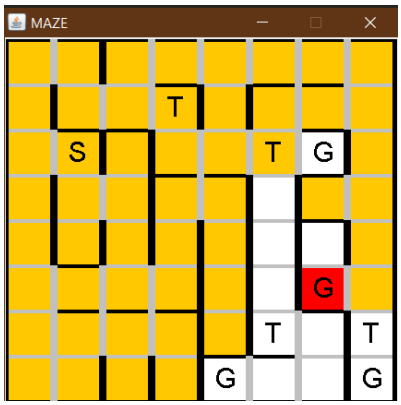


After this process, the start window will appear again. The result of the algorithm is shown in the output.txt as follows:

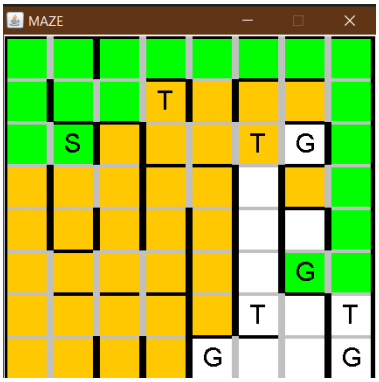
```
================================================================================
Date 04/12/2019 01:46:50
Iterative Deepening

Solution Path :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (1, 3) => (2, 3) =>
(2, 4) => (2, 5) => (2, 6)

The List of Expanded Nodes :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (0, 2) => (0, 3) =>
(0, 4) => (1, 4) => (2, 4) => (0, 5) => (0, 6) => (1, 3) => (2, 3) => (2, 5) => (1, 5) =>
(3, 5) => (2, 6)

The cost : 23

================================================================================
```

3.5. Uniform Cost Search

When we choose Uniform Cost Search, the maze window will appear as follows:



The way the solution is searched will proceed as in Depth First Search. If the algorithm finds the solution, squares in the solution path will be painted in green.

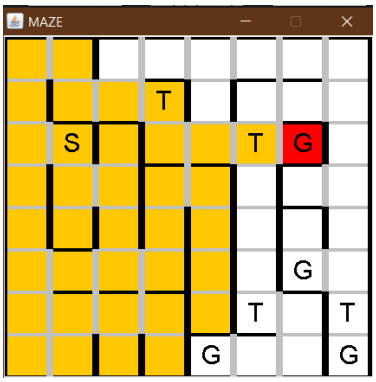


After this process, the start window will appear again. The result of the algorithm is shown in the output.txt as follows:

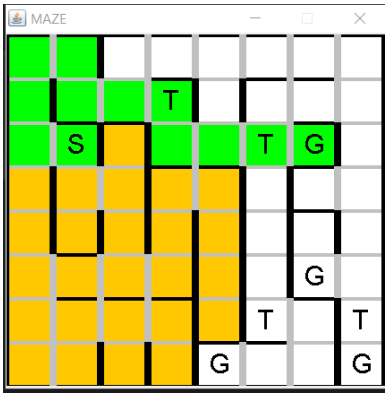
```
==========================================================================
Date 04/12/2019 01:47:19
Uniform Cost Search

Solution Path :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (0, 2) => (0, 3) =>
(0, 4) => (0, 5) => (0, 6) => (0, 7) => (1, 7) => (2, 7) => (3, 7) => (4, 7) => (5, 7) =>
(5, 6)

The List of Expanded Nodes :
(2, 1) => (3, 1) => (2, 0) => (2, 0) => (1, 0) => (3, 0) => (4, 1) => (3, 2) => (4, 0) =>
(2, 2) => (0, 0) => (4, 2) => (0, 1) => (5, 2) => (5, 0) => (5, 1) => (6, 0) => (1, 1) =>
(5, 1) => (5, 3) => (5, 0) => (4, 3) => (6, 1) => (7, 0) => (5, 0) => (1, 2) => (3, 3) =>
(0, 2) => (7, 1) => (6, 2) => (7, 0) => (7, 2) => (6, 3) => (3, 4) => (0, 3) => (4, 4) =>
(0, 4) => (7, 3) => (1, 4) => (0, 5) => (5, 4) => (0, 6) => (6, 4) => (2, 4) => (0, 7) =>
(2, 3) => (2, 3) => (1, 3) => (1, 7) => (2, 7) => (1, 6) => (1, 6) => (1, 5) => (1, 5) =>
(1, 5) => (3, 7) => (4, 7) => (3, 6) => (3, 6) => (5, 7) => (2, 5) => (5, 6)

The cost : 18

==========================================================================================
```

3.6. Greedy Best First Search

When we choose Greedy Best First Search, the maze window will appear as follows:



The way the solution is searched will proceed as in Depth First Search. If the algorithm finds the solution, squares in the solution path will be painted in green.

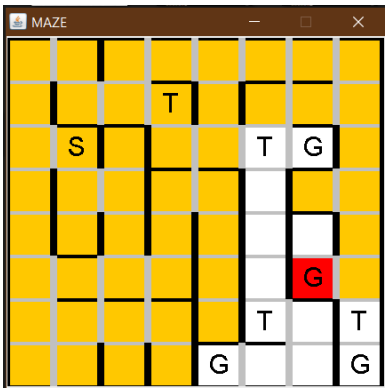


After this process, the start window will appear again. The result of the algorithm is shown in the output.txt as follows:

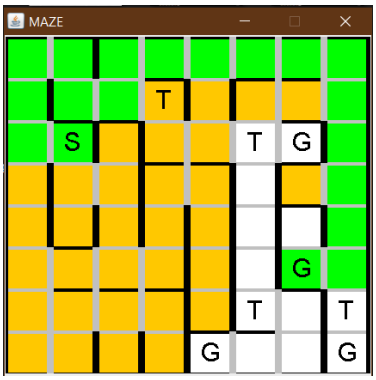
```
=============================================================
Date 04/12/2019 01:47:42
Greedy Best First Search

Solution Path :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (1, 3) => (2, 3) =>
(2, 4) => (2, 5) => (2, 6)

The List of Expanded Nodes :
(2, 1) => (3, 1) => (3, 2) => (2, 2) => (4, 2) => (5, 2) => (5, 3) => (4, 3) => (3, 3) =>
(3, 4) => (4, 4) => (5, 4) => (6, 4) => (5, 1) => (5, 1) => (5, 0) => (6, 0) => (6, 1) =>
(6, 2) => (6, 3) => (7, 3) => (7, 2) => (7, 1) => (7, 0) => (7, 0) => (5, 0) => (4, 1) =>
(5, 0) => (2, 0) => (2, 0) => (1, 0) => (3, 0) => (4, 0) => (0, 0) => (0, 1) => (1, 1) =>
(1, 2) => (1, 3) => (2, 3) => (2, 4) => (2, 5) => (2, 6)

The cost : 23

=============================================================
```

3.7. A* Heuristic Search

When we choose A* Heuristic Search, the maze window will appear as follows:



The way the solution is searched will proceed as in Depth First Search. If the algorithm finds the solution, squares in the solution path will be painted in green.



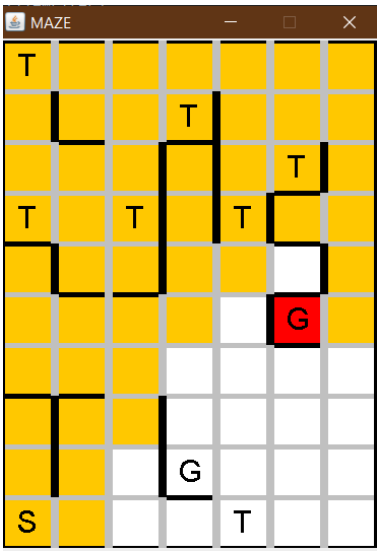
After this process, the start window will appear again. The result of the algorithm is shown in the output.txt as follows:

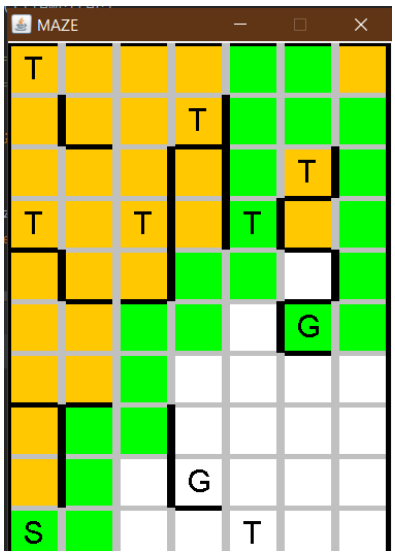
```
==========================================================================
Date 04/12/2019 01:48:14
A* Heuristic Search

Solution Path :
(2, 1) => (2, 0) => (1, 0) => (0, 0) => (0, 1) => (1, 1) => (1, 2) => (0, 2) => (0, 3) =>
(0, 4) => (0, 5) => (0, 6) => (0, 7) => (1, 7) => (2, 7) => (3, 7) => (4, 7) => (5, 7) =>
(5, 6)

The List of Expanded Nodes :
(2, 1) => (3, 1) => (2, 0) => (2, 0) => (3, 2) => (2, 2) => (4, 2) => (4, 1) => (5, 2) =>
(5, 3) => (3, 0) => (1, 0) => (4, 0) => (5, 1) => (5, 0) => (5, 1) => (6, 0) => (4, 3) =>
(6, 1) => (7, 0) => (7, 1) => (6, 2) => (6, 3) => (7, 2) => (7, 3) => (3, 3) => (0, 0) =>
(3, 4) => (0, 1) => (1, 1) => (1, 2) => (4, 4) => (7, 0) => (5, 4) => (5, 0) => (6, 4) =>
(5, 0) => (0, 2) => (0, 3) => (0, 4) => (0, 5) => (1, 4) => (0, 6) => (2, 4) => (0, 7) =>
(2, 3) => (1, 7) => (2, 3) => (2, 7) => (1, 6) => (1, 6) => (1, 5) => (1, 5) => (1, 5) =>
(1, 3) => (3, 7) => (3, 6) => (3, 6) => (4, 7) => (5, 7) => (5, 6)

The cost : 18

==========================================================================================
```

4. Execution with a Different Maze

We used another maze input to test our algorithms. The results are obtained with this maze as listed below.

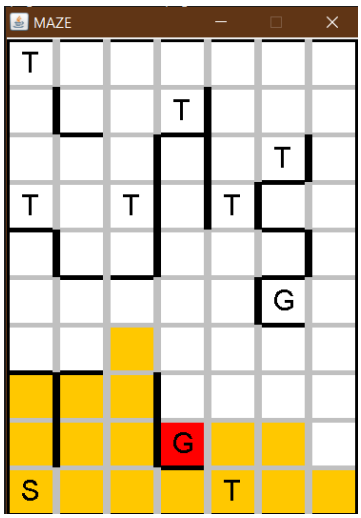
4.1. Depth-First Search

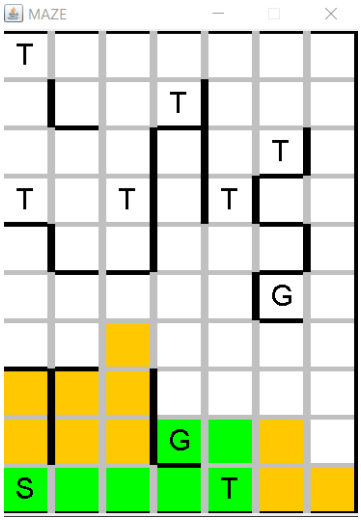





```
============================================================
Date 04/12/2019 18:52:54
Depth First Search

Solution Path :
(9, 0) => (9, 1) => (8, 1) => (7, 1) => (7, 2) => (6, 2) => (5, 2) => (5, 3) => (4, 3) =>
(4, 4) => (3, 4) => (2, 4) => (1, 4) => (0, 4) => (0, 5) => (1, 5) => (1, 6) => (2, 6) =>
(3, 6) => (4, 6) => (5, 6) => (5, 5)

The List of Expanded Nodes :
(9, 0) => (8, 0) => (7, 0) => (9, 1) => (8, 1) => (7, 1) => (7, 2) => (6, 2) => (5, 2) =>
(5, 1) => (5, 0) => (4, 0) => (6, 0) => (6, 1) => (5, 3) => (4, 3) => (3, 3) => (2, 3) =>
(4, 4) => (3, 4) => (2, 4) => (1, 4) => (0, 4) => (0, 3) => (0, 2) => (0, 1) => (0, 0) =>
(1, 0) => (2, 0) => (3, 0) => (3, 1) => (2, 1) => (2, 2) => (1, 2) => (1, 1) => (1, 3) =>
(3, 2) => (4, 2) => (4, 1) => (0, 5) => (1, 5) => (2, 5) => (1, 6) => (0, 6) => (2, 6) =>
(3, 6) => (3, 5) => (4, 6) => (5, 6) => (5, 5)

The cost : 27

================================================================================
```

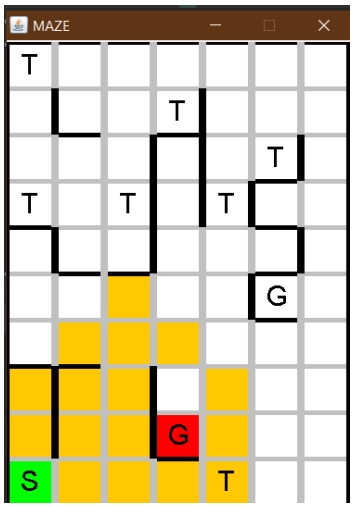
4.2. Breadth-First Search

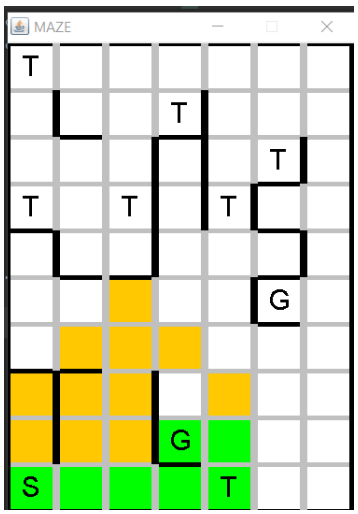





```
================================================================
Date 04/12/2019 18:53:31
Breadth First Search

Solution Path :
(9, 0) => (9, 1) => (9, 2) => (9, 3) => (9, 4) => (8, 4) => (8, 3)

The List of Expanded Nodes :
(9, 0) => (9, 1) => (8, 0) => (9, 2) => (8, 1) => (7, 0) => (9, 3) => (8, 2) => (7, 1) =>
(9, 4) => (8, 1) => (7, 2) => (9, 5) => (8, 4) => (7, 1) => (6, 2) => (9, 6) => (8, 5) =>
(8, 3)

The cost : 12

================================================================================================
```

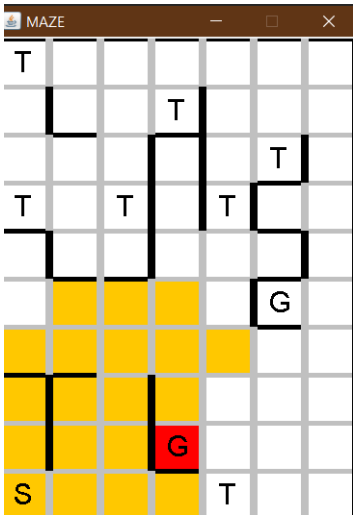
4.3. Iterative Deepening

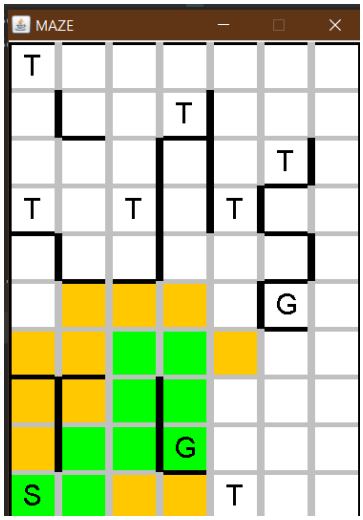





```
================================================================
Date 04/12/2019 18:56:05
Iterative Deepening

Solution Path :
(9, 0) => (9, 1) => (9, 2) => (9, 3) => (9, 4) => (8, 4) => (8, 3)

The List of Expanded Nodes :
(9, 0) => (8, 0) => (7, 0) => (9, 1) => (8, 1) => (7, 1) => (7, 2) => (6, 2) => (5, 2) =>
(6, 1) => (6, 3) => (8, 2) => (9, 2) => (9, 3) => (9, 4) => (8, 4) => (7, 4) => (8, 3)

The cost : 12

================================================================================================
```

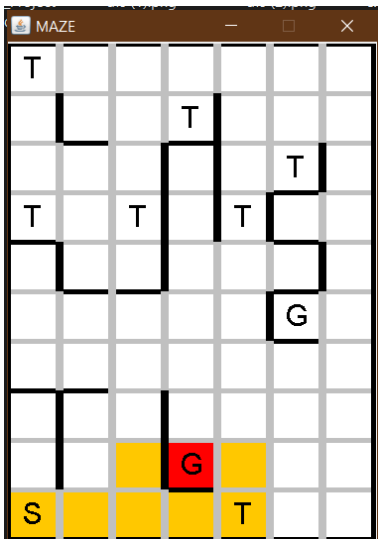
4.4. Uniform Cost Search

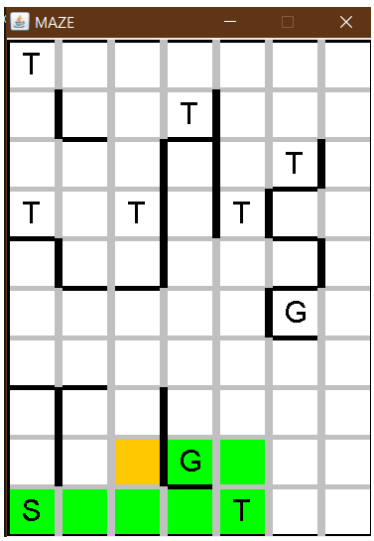





```
================================================================
Date 04/12/2019 18:57:10
Uniform Cost Search

Solution Path :
(9, 0) => (9, 1) => (8, 1) => (8, 2) => (7, 2) => (6, 2) => (6, 3) => (7, 3) => (8, 3)

The List of Expanded Nodes :
(9, 0) => (9, 1) => (8, 0) => (8, 1) => (7, 0) => (9, 2) => (8, 2) => (9, 3) => (7, 1) =>
(7, 2) => (7, 1) => (6, 2) => (6, 3) => (5, 2) => (6, 1) => (6, 1) => (6, 0) => (6, 0) =>
(5, 3) => (5, 1) => (6, 0) => (6, 4) => (7, 3) => (5, 1) => (8, 3)

The cost : 8

================================================================================================
```

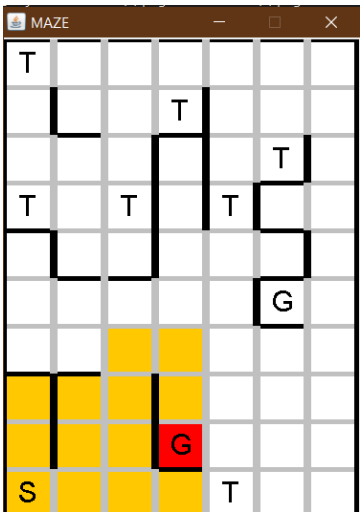
4.5. Greedy Best First Search

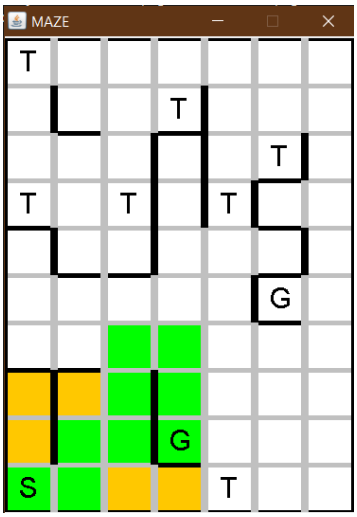




```
================================================================
Date 04/12/2019 18:57:39
Greedy Best First Search

Solution Path :
(9, 0) => (9, 1) => (9, 2) => (9, 3) => (9, 4) => (8, 4) => (8, 3)

The List of Expanded Nodes :
(9, 0) => (9, 1) => (9, 2) => (9, 3) => (8, 2) => (9, 4) => (8, 4) => (8, 3)

The cost : 12

================================================================================================
```

4.6. A* Heuristic Search






```
================================================================
Date 04/12/2019 18:58:24
A* Heuristic Search

Solution Path :
(9, 0) => (9, 1) => (8, 1) => (8, 2) => (7, 2) => (6, 2) => (6, 3) => (7, 3) => (8, 3)

The List of Expanded Nodes :
(9, 0) => (9, 1) => (8, 0) => (8, 1) => (9, 2) => (8, 2) => (9, 3) => (7, 2) => (7, 0) =>
(7, 1) => (7, 1) => (6, 2) => (6, 3) => (7, 3) => (8, 3)

The cost : 8

================================================================================================
```

5. Comparison of Algorithms and Conclusion

Search Algorithm	Maze 1		Maze 2	
	Solution Path Cost	Expanded Square Count	Solution Path Cost	Expanded Square Count
Depth First Search	26	27	27	50
Breadth First Search	23	47	12	19
Iterative Deepening Search	23	47	12	19
Uniform Cost Search	18	62	8	25
Greedy Best First Search	23	42	12	8
A* Heuristics Search	18	61	8	15

As a result, some search algorithms have produced similar results. If we analyze the path cost results of the algorithms for both maze samples, the Depth First Search algorithm gave the worst result because it searched until the end of the selected branch.

The Iterative Deepening algorithm is similar to the Breadth-First Search algorithm, as it progresses to depth by depth on DFS. Therefore, the output of these two algorithms in parallel. In the Iterative Deepening, the process took a little longer because it tried to find the closest goal. As seen in the GUI, the search process was like trying to find the nearest goal by circling the maze at every depth without paying any attention to the costs.

The Greedy Best First Algorithm waived the cost to find the nearest goal node. Finally, Uniform Cost Search and A* Heuristics Search algorithms have become the most ideal algorithms because they pay attention to the cost of extended squares and the Manhattan distance to the nearest goal.