ARTIFICIAL INTELLIGENCE
CSL-325

Project: A MAZE SOLVING PROGRAM

Group Member Name

- **Daniyal Qureshi (02-134191-055)**
- **Salman Siddiqui (02-134191-007)**
- **Faraz Ali (02-134191-114)**

COUSE INTRUCTOR: **AMNA IFTIKHAR**
LAB INSTRUCTOR: **SALAS AKBAR**

BAHRIA UNIVERSITY, KARACHI, PAKISTAN

DEPARTMENT OF COMPUTER SCIENCE

Table of Contents

ACKNOWLEDGMENT:

Team effort together with precious words of encouragement and guidance makes daunting tasks achievable. It is a pleasure to acknowledge the direct and implied help we have received at various stages in the task of developing the project. It would not have been pos sible to develop such a project without the furtherance on part of numerous individuals. We find it impossible to express our thanks to each one of them in words, for it seems too trivial when compare to the profound encouragement that they extended to us

We would like to express our gratitude towards our instructors for their kind co-operation and encouragement which help us in completion of this project. We would like to express our special gratitude and thanks to Miss Salas Akbar for giving us such attention and time. Completion of this project could not have been accomplished without the support of our course instructor and lab instructor, Miss Amna Iftikhar and Miss Salas Akbar.

CHAPTER 01**: INTRODUCTION**

1.1 INTRODUCTION TO SYSTEM:

Maze solving program uses the breadth first search (BFS) algorithm. BFS is more effective algorithm because it searches all the paths at the same time unlike the depth first search (DFS) whose only searches one path at a time. Once it has identified all the cells in the maze it will start up backtrack using the most efficient route. When it gets to the purple spot it will use the most effective route back to the red spot which is the starting position. It uses two things frontier list and a visited list to store the information and a variable called content. Backtracking uses a python dictionary which consist of two key elements (key and value). It is used to identify the cell in which the current cell is and the cell that came from. This is a demonstration of a maze solving program using the breadth first search algorithm. In this report, genetic algorithms were used BFS and Backtracking for solving mazes with regards to computational time and solution. This was done by creating mazes. Hence the BFS is the most effective algorithm because it searches all the paths at the same time unlike the depth first search which only searches one path at a time. That's why we implemented and chose this algorithm in our maze. Once it has identified all the cells in the maze it will start up the backtracking using the most efficient route. Maze solving is the act of finding a route or path through the maze from start to finish. Some maze solving methods are designed to be used inside the maze by a traveler with no prior knowledge of the maze, whereas others are designed to be used by a person or computer program that can see the whole maze at once.

1.2 Background of the system:

Robotic field has caught a big attention to our world as it conveniently increases efficiency of especially monotonous work. This field does include the industrial robots and mobile robots that is used for various purposes. As the fields become open for everyone, a lot of competition such as Micro Mouse competition started to build up as to making the best robot. In the Micro Mouse competition, it normally uses 256 square units of maze. Micro mice or the robot will compete to solve the maze without the needs of any manual assistance. In order to do so, suitable algorithm should be used to solve the maze in the least time possible.

1.3 Motivation:

We encounter many different kinds of puzzles every day and often need the shortest path between two points or a solution. Our motivation behind this project was to create a program which can automatically solve a maze for the user. Maze solving and shortest path algorithms within image processing are very important in a number of different applications ranging from route mapping to feature extraction and seam carving within images

1.4 Methodology:

Mazes are often simple puzzles for humans, but they present a great programming problem that we can solve using shortest-path techniques like BFS. BFS uses the opposite strategy as depth-first search (DFS), which instead explores the node branch

as far as possible before being forced to backtrack. Still, both algorithms have the same lines of code & the only difference is the data structure used in it. In short we can say that this is an incredibly useful algorithm that guarantees the shortest path between two nodes. Breadth First Search explores equally in all directions until the goal is reached. In other words, it starts from a chosen node and examine all its neighbors, then it examines all the neighbors of the neighbors, and so on!

1.5 Brief System Description:

Mazes in general are extremely varied in terms of complexity, topology, and shape, so we first need to restrict our space of mazes to those which are simply-connected, meaning there are only two disjoint walls in our maze, and those which have their starts and ends on their edge. Note that within this definition we do not require a strictly rectangular maze, but that for the most part, rectangular shaped mazes will be our primary type of maze by default. This is a demonstration of a maze solving program using the breadth first search algorithm. In this report, genetic algorithms were used BFS and Backtracking for solving mazes with regards to computational time and solution path length. This was done by creating mazes. Hence the BFS is the most effective algorithm because it searches all the paths at the same time unlike the depth first search which only searches one path at a time. That's why we implemented and chose this algorithm in our maze. Once it has identified all the cells in the maze it will start up the backtracking using the most efficient route.

1.6 Tools Used / System Specification:

Import functions:

TURTLE: Import turtle allows us to use the turtle library in python. Turtle allows us to draw on the screen and it is based upon an old programming language called turtle. Turtle is a pre-installed python library that enables users to create pictures and shapes by providing them with a virtual canvas. The onscreen pen that you use for drawing is called the turtle.

TIME: There is a popular time module available in python which provides functions for working with times, and for converting between representations. The function time returns the current system time in ticks.

SYS: It lets us access system-specific parameters and functions. First, we have to import the sys module in our program before running any functions. This function provides the name of the existing python modules which have been imported. This function returns a list of command line arguments to a python script

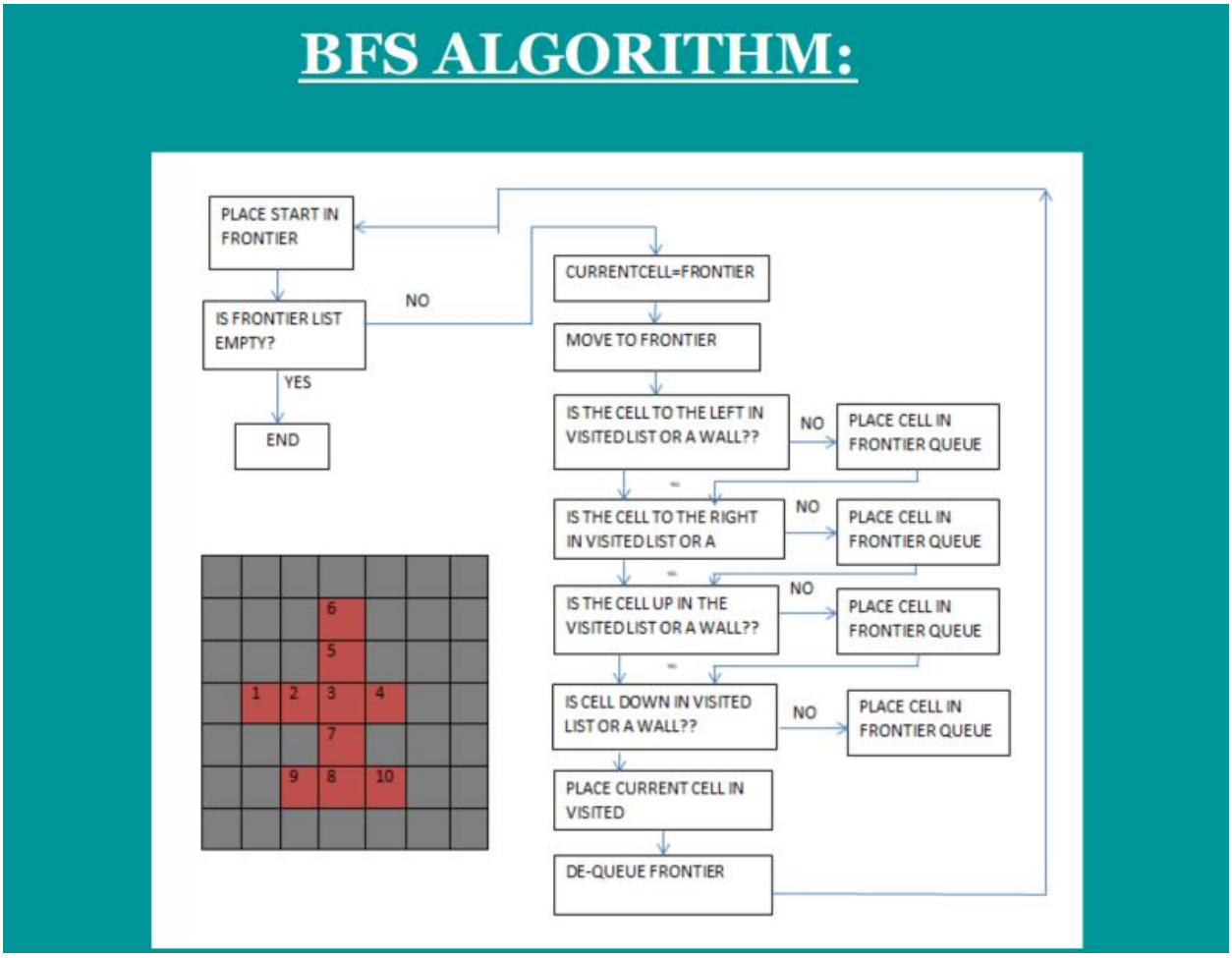
CHAPTER 02**: Literature Review**

2.1 Literature Review:

The most crucial thing to be decided in this project is the algorithm that it will be using to navigate through and solve the maze. In determining the algorithms, the most efficient and effective method must be chosen. As each of the method proposed has their own pros and cons, finding the method with least disadvantages may help in making this project a successful project. Following method may took time in determining its route, it is the convenient way of finding the destination set in the maze BFS uses nodes as it paths in determining the shortest route to the destination. An algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level.

CHAPTER 02**: Designed System**

2.2 Algorithms Implementation with Explanation:



BACKTRACKING ALGORITHM:

Basically it uses a key feature of python list which is called as a Dictionary. Which consists of two key elements so you have the solution which is the list name and you have got two values, you have a key. So the key returns the value. I am using this to identify the cell that I am on and the cell that I came from. For instance if I take cell 1 built the cell 2 is the current cell that previously come from cell 1.

CURRENT (KEY)	PREVIOUS (VALUE)
1	1
2	1
3	2
4	3
5	3
6	5
7	3
8	7
9	8
10	8

CHAPTER 02: **Results and Conclusion:**

2.3 Comparative Analysis:

We have found many algorithm like pledge algorithm, wall follower and random mouse algorithm but BFS and backtracking are best for this project because Breadth-**first search** (**BFS**) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored. For example, in a chess endgame a chess engine may build the game tree from the current position by applying all possible moves, and use breadth-first search to find a win position for white. Implicit trees (such as game trees or other problem-solving trees) may be of infinite size; breadth-first search is guaranteed to find a solution node[1] if one exists. Backtracking is a **technique based on algorithm to solve problem**. It uses recursive calling to find the solution by building a solution step by step increasing values with time. It removes the solutions that doesn't give rise to the solution of the problem based on the constraints given to solve the problem.

2.4 Conclusion:

If you correctly understand the steps cited in this report, you'll be able to fully understand the codebase I wrote from scratch to implement all of this. Mazes are problems with obstacles, a start node and an end node. The solution to the maze is the path from start node to the end node avoiding all the obstacles. Path finding algorithms can be used to solve the mazes which in our case we have used breadth first search (BFS). Since path finding generally refers to find the shortest route between two end points. Various algorithms have been developed to find the most efficient path in the minimum time between nodes. The breadth first search algorithm uses a queue to visit cells in increasing. distance order distance order from the start until the finish is reached. Each visited cell needs to keep track of its distance from the start caused it to be added to the queue.

CHAPTER 02: **Future Work and Conclusion:**

2.5 Problems Faced:

The problem faced is that whether the algorithm will be using wall-following, flood-filling, BFS search algorithm and many more method possible or whether to combine two or more algorithm in a project. There are pro and cons for every each of the algorithm that may varies the results in different efficiency and effectiveness in solving the maze. Other consideration is that how much time will it take to find the destination as the robot itself will take time to process it's surrounding depending on the algorithm being used by the robot.

2.6 Future Work:

Basically this Maze solver can be used to make Applications systems including intelligent traffic control that helps ambulances, fire fighters, or rescuing robots to find their shortest path to their destination.

2.7 Code:

```
import turtle               # import turtle library
import time
import sys
from collections import deque

wn = turtle.Screen()            # define the turtle screen
wn.bgcolor("black")             # set the background colour
wn.title("MAZE SOLVER USING BFS & BACKTRACKING")
wn.setup(1200,660)              # setup the dimensions of the working window


# this is the class for the Maze
class Maze(turtle.Turtle):            # define a Maze class
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")          # the turtle shape
        self.color("gold")            # colour of the turtle
        self.penup()                # lift up the pen so it do not leave a trail / Positional argument
        self.speed(0)

# this is the class for the finish line - green square in the maze
class Green(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("turtle")
        self.color("green")
```

```
        self.penup()
        self.speed(0)

class Cyan(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("blue")
        self.penup()
        self.speed(0)


# this is the class for the yellow or turtle
class Red(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("arrow")
        self.color("red")
        self.penup()
        self.speed(0)

class Crimson(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("circle")
        self.color("crimson")
        self.penup()
        self.speed(0)

grid1= [
"++++++++++++++++",
"+s+      + +e+",
"+ +++++ +++ + +",
"+ + +      + +",
"+ +   +++ + + +",
"+ + + +   + + +",
"+   + +   + + +",
"+++++ +   + + +",
"+     +   +   +",
"++++++++++++++++",
]

grid2= [
"+++++++++",
```

```
"+ ++s++++",
"+ ++ ++++",
"+ ++ ++++",
"+   ++++",
"++++ ++++",
"++++ ++++",
"+    e+",
"+++++++++",
]

grid3= [
"+++++++++++++++",
"+       e  +",
"+         +",
"+         +",
"+         +",
"+         +",
"+         +",
"+         +",
"+ s        +",
"+++++++++++++++",
]

grid4= [
"++++++++++++++++++++++++++++++++++++++++++++++++++++++++++",
"+        ++                    +",
"+ ++++++++++ ++++++++++++ +++++++ ++++++++++++",
"+      +          +         ++  +",
"+ +++++++ ++++++++++++ ++++++++++++++++++++ +",
"+ +   + +      + +         s +++ +",
"+ + + + + + ++++ + + ++++++++++++ +++ +",
"+ + + + + + +     + + +     +    +",
"+ + ++++ + ++++++++++ + + ++++ + + ++  +",
"+ +   + +      +  +      + + ++ ++",
"+ ++++ + +++++++ ++++++++ ++++++++++++ ++ ++",
"+   + +   +        +        ++  +",
"++++ + ++++++++++ ++++++++++ ++++++++++ +++ +",
"+ + +           +   +   + + +++ +",
"+ + ++++ ++++++++++++ + ++++ + + + ++  +",
"+ + +   +   +   + + +   +   + ++ ++",
"+ + + +++++++ ++++ + + + ++++++++++ ++ ++",
"+             + + +        ++ ++",
"+ ++++++       + + + + +++     +++ ++",
"+ ++++++ ++++++ +++++++++   ++ ++  ++++++++++ ++",
```

```
"+ +   +   +++ +   +++++++++ ++ +++++++   + ++",
"+ ++++ ++++ +++ + +++ +++   ++   ++ e ++ ++ + ++",
"+ ++++   +   + +++ +++ ++ ++++++++ ++ ++ ++  ++",
"+    ++ ++++++++++   ++      ++  +++++++",
"++++++++++++++++++++++++++++++++++++++++++++++++++",
]

grid5= [
"++++++++++++++++++++++++++++++++++++++++++++++++++",
"+        e+                   +",
"++++++++++++++++++++++++++++++++++++++++++++++++++",
"+      +          +         ++  +",
"+ +++++++ ++++++++++++ ++++++++++++++++++++++++++",
"+ +   + +      + +          +++ +",
"+ + + + + + ++++ + + ++++++++++++ +++ +",
"+ + + + + + +    + + +    +    +",
"+ + ++++ + ++++++++++ + + ++++ + + ++  +",
"+ +   + +      +  +      + + ++ ++",
"+ ++++ + +++++++ ++++++++ ++++++++++++ ++ ++",
"+   + +   +        +        ++  +",
"++++ + ++++++++++ ++++++++++ ++++++++++ +++ +",
"+ + +           +   +   + + +++ +",
"+ + ++++ ++++++++++++ + ++++ + + + ++  +",
"+ + +   +   +   + + +   +   + ++ ++",
"+ + + +++++++ ++++ + + + ++++++++++ ++ ++",
"+            + + +        ++ ++",
"+ ++++++       + + + + +++     +++ ++",
"+ ++++++ ++++++ +++++++++  ++ ++  ++++++++++ ++",
"+ +   +   +++ +   +++++++++ ++ +++++++   + ++",
"+ ++++ ++++ +++ + +++ +++   ++   ++   ++ ++ + ++",
"+ ++++   +   + +++ +++ ++ ++++++++ ++ ++ ++  ++",
"+    ++ ++++++++++   ++      ++  s+++++++",
"++++++++++++++++++++++++++++++++++++++++++++++++++",
]

#grid6= [
#"++++++++++++++++++++++++++++++++++++++++++++++++++",
#"+       +                   +",
#"++++++++++++++++++++++++++++++++++++++++++++++++++",
#"+      +          +         ++  +",
#"+ +++++++ ++++++++++++ ++++++++++++++++++++++++++",
#"+ +   + +      + +          +++ +",
#"+ + + + + + ++++ + + ++++++++++++ +++ +",
#"+ + + + + + +    + + +    +    +",
```

```
#"+ + ++++ + ++++++++++ + + ++++ + + ++  +",
#"+ +   + +     +  +      + + ++ ++",
#"+ ++++ + +++++++ ++++++++ ++++++++++++ ++ ++",
#"+   + +   +        +       ++  +",
#"++++ + ++++++++++ ++++++++++ ++++++++++ +++ +",
#"+ + +           +   +   + + +++ +",
#"+ + ++++ ++++++++++++ + ++++ + + + ++  +",
#"+ + +   +   +   + + +   +   + ++ ++",
#"+ + + +++++++ ++++ + + + ++++++++++ ++ ++",
#"+             + + +     es ++ ++",
#"+ ++++++       + + + + +++     +++ ++",
#"+ ++++++ ++++++ +++++++++   ++ ++  ++++++++++ ++",
#"+ +   +   +++ +   +++++++++ ++ +++++++   + ++",
#"+ ++++ ++++ +++ + +++ +++   ++   ++   ++ ++ + ++",
#"+ ++++   +   + +++ +++ ++ ++++++++ ++ ++ ++  ++",
#"+    ++ ++++++++++++   ++      ++   +++++++",
#"++++++++++++++++++++++++++++++++++++++++++++++++++",
# ]

#grid7= [
#"++++++++++++++++",
#"+ ++++ +e+",
#"+ +++++ +++ + +",
#"+ + +     + +",
#"+ +  +++ + + +",
#"+ + + +  + + +",
#"+  + +  + + +",
#"+++++ +  + + +",
#"+   +  +  +",
#"++++++++++++++++",
#]

#grid7= [
#"++++++++++++++++",
#"+ ++++ +s+",
#"+ +++++ +++ + +",
#"+ + +     + +",
#"+ +++++ + + +",
#"+ + + +  + + +",
#"+  + +  + + +",
#"+++++ +  + + +",
#"+   +  +  +",
#"++++++++++++++++",
#]
```

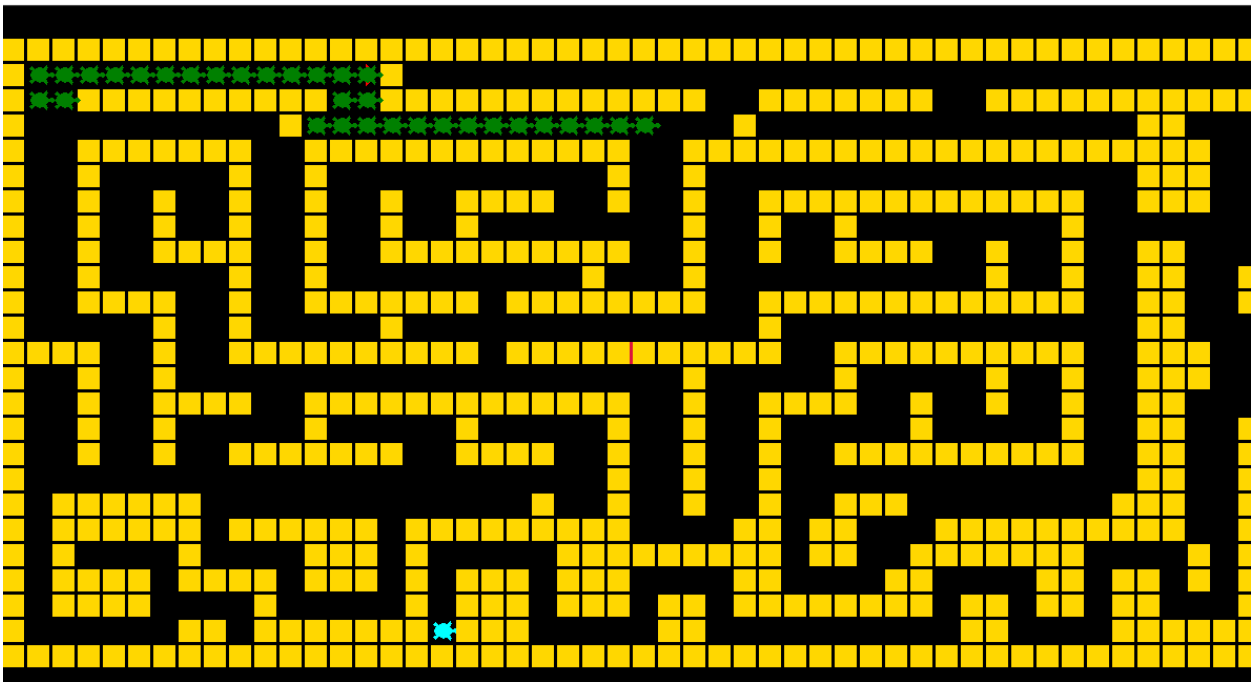
```
grid= [
"++++++++++++++++++++++++++++++++++++++++++++++++++++",
"+        s+                    +",
"+ ++++++++++ ++++++++++++ +++++++ ++++++++++++",
"+      +          +         ++  +",
"+ +++++++ ++++++++++++ ++++++++++++++++++++ +",
"+ +   + +       + +          +++ +",
"+ + + + + + ++++ + + ++++++++++++ +++ +",
"+ + + + + + +     + + +     +    +",
"+ + ++++ + ++++++++++ + + ++++ + + ++  +",
"+ +   + +      +  +       + + ++ ++",
"+ ++++ + +++++++ ++++++++ ++++++++++++ ++ ++",
"+   + +   +        +        ++  +",
"++++ + ++++++++++ ++++++++++ ++++++++++ +++ +",
"+ + +           +   +   + + +++ +",
"+ + ++++ ++++++++++++ + ++++ + + + ++  +",
"+ + +   +   +   + + +   +   + ++ ++",
"+ + + +++++++ ++++ + + + ++++++++++ ++ ++",
"+             + + +        ++ ++",
"+ ++++++        + + + + +++     +++ ++",
"+ ++++++ ++++++ +++++++++   ++ ++  ++++++++++ ++",
"+ +   +   +++ +   +++++++++ ++ +++++++   + ++",
"+ ++++ ++++ +++ + +++ +++   ++   ++   ++ ++ + ++",
"+ ++++   +   + +++ +++ ++ ++++++++ ++ ++ ++  ++",
"+    ++ +++++++e+++   ++      ++  +++++++",
"++++++++++++++++++++++++++++++++++++++++++++++++++++",
 ]


def setup_maze(grid):                    # define a function called setup_maze
   global start_x, start_y, end_x, end_y     # set up global variables for start and end locations
   for y in range(len(grid)):             # read in the grid line by line
      for x in range(len(grid[y])):        # read each cell in the line
         character = grid[y][x]           # assign the varaible "character" the the x and y location od
the grid
         screen_x = -588 + (x * 24)       # move to the x location on the screen staring at -588
         screen_y = 288 - (y * 24)        # move to the y location of the screen starting at 288

         if character == "+":
            maze.goto(screen_x, screen_y)        # move pen to the x and y locaion and
            maze.stamp()                         # stamp a copy of the turtle on the screen
            walls.append((screen_x, screen_y))    # add coordinate to walls list
```

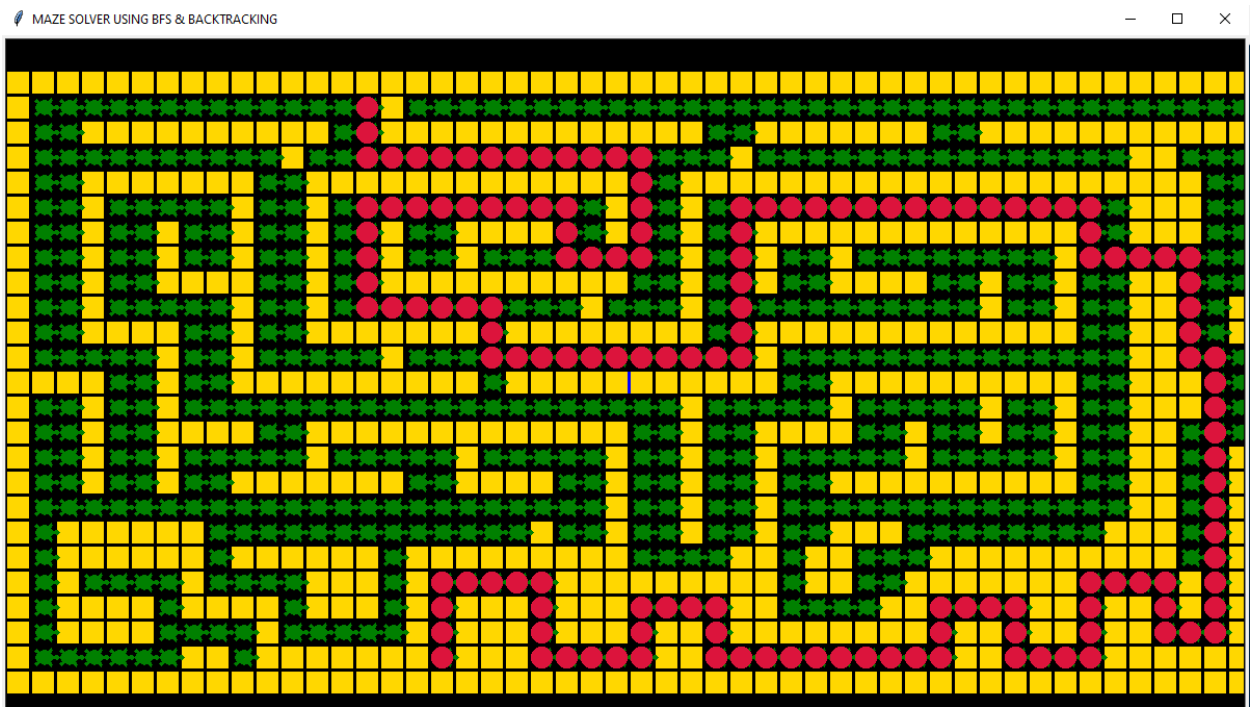
```
        if character == " " or character == "e":
            path.append((screen_x, screen_y))     # add " " and e to path list

        if character == "e":
            green.color("cyan")
            green.goto(screen_x, screen_y)       # send green sprite to screen location
            end_x, end_y = screen_x,screen_y     # assign end locations variables to end_x and
end_y
            green.stamp()
            green.color("green")

        if character == "s":
            start_x, start_y = screen_x, screen_y  # assign start locations variables to start_x and
start_y
            red.goto(screen_x, screen_y)


def endProgram():
    wn.exitonclick()
    sys.exit()

def search(x,y):                #bfs code starts here
    frontier.append((x, y))
    solution[x,y] = x,y

    while len(frontier) > 0:         # exit while loop when frontier queue equals zero
        time.sleep(0)
        x, y = frontier.popleft()     # pop next entry in the frontier queue an assign to x and y
location

        if(x - 24, y) in path and (x - 24, y) not in visited:  # check the cell on the left
            cell = (x - 24, y)
            solution[cell] = x, y    # backtracking routine [cell] is the previous cell. x, y is the current
cell
            #blue.goto(cell)       # identify frontier cells
            #blue.stamp()
            frontier.append(cell)   # add cell to frontier list
            visited.add((x-24, y))  # add cell to visited list

        if (x, y - 24) in path and (x, y - 24) not in visited:  # check the cell down
            cell = (x, y - 24)
            solution[cell] = x, y
            #blue.goto(cell)
            #blue.stamp()
```

```
            frontier.append(cell)
            visited.add((x, y - 24))
            print(solution)

        if(x + 24, y) in path and (x + 24, y) not in visited:   # check the cell on the  right
            cell = (x + 24, y)
            solution[cell] = x, y
            #blue.goto(cell)
            #blue.stamp()
            frontier.append(cell)
            visited.add((x +24, y))

        if(x, y + 24) in path and (x, y + 24) not in visited:  # check the cell up
            cell = (x, y + 24)
            solution[cell] = x, y
            #blue.goto(cell)
            #blue.stamp()
            frontier.append(cell)
            visited.add((x, y + 24))
        green.goto(x,y)
        green.stamp()


def backRoute(x, y):            #Backtracking starts here
    yellow.goto(x, y)
    yellow.stamp()
    while (x, y) != (start_x, start_y):    # stop loop when current cells == start cell
        yellow.goto(solution[x, y])       # move the yellow sprite to the key value of solution ()
        yellow.stamp()
        x, y = solution[x, y]             # "key value" now becomes the new key

# set up classes
maze = Maze()
red = Red()
blue = Cyan()
green = Green()
yellow = Crimson()

# setup lists
walls = []
path = []
visited = set()
frontier = deque()
solution = {}                    # solution dictionary
```

```
# main program starts here ####
setup_maze(grid)
search(start_x,start_y)
backRoute(end_x, end_y)
wn.exitonclick()
```

2.8 Screenshots:





Chapter References

1. References

2. Maze to Tree on YouTube
3. ^ Maze Transformed on YouTube
4. ^ *Abelson; diSessa (1980), Turtle Geometry: the computer as a medium for exploring mathematics, ISBN 9780262510370*
5. ^ Seymour Papert, "Uses of Technology to Enhance Education", *MIT Artificial Intelligence Memo No. 298*, June 1973
6. ^ Public conference, December 2, 2010 – by professor Jean Pelletier-Thibert in Academie de Macon (Burgundy – France) – (Abstract published in the Annals academic, March 2011 – ISSN 0980-6032) Charles Tremaux (° 1859 – † 1882) Ecole Polytechnique of Paris (X:1876), French engineer of the telegraph
7. ^ Édouard Lucas: *Récréations Mathématiques* Volume I, 1882.
8. ^ H. Fleischner: *Eulerian Graphs and related Topics.* In: *Annals of Discrete Mathematics* No. 50 Part 1 Volume 2, 1991, page X20.
9. ^ *Even, Shimon (2011), Graph Algorithms (2nd ed.), Cambridge University Press, pp. 46–48, ISBN 978-0-521-73653-4*.
10. ^ *Sedgewick, Robert (2002), Algorithms in C++: Graph Algorithms (3rd ed.), Pearson Education, ISBN 978-0-201-36118-6*.

11. ^ *Fattah, Mohammad; et, al. (2015-09-28). "A Low-Overhead, Fully-Distributed, Guaranteed-Delivery Routing Algorithm for Faulty Network-on-Chips". NOCS '15 Proceedings of the 9th International Symposium on Networks-on-Chip. Nocs '15: 1–8. doi:10.1145/2786572.2786591. ISBN 9781450333962. S2CID 17741498*