(The heuristics are discussed on page 8 for DepthFirstSearch and inside the relative code commented by # and on page 18 for Breadth First Search and inside the relative code commented by #)

Report: Maze Game (Depth First Search)

To avoid confusion I merged the two algorithms presented in the previous Homework into a single file where you can choose which algorithm to start by commenting out the line of code of one and uncommenting the other. For this exercise, I used the Python turtle library which provides the primitives of the turtle graphics in both object-oriented and procedural form, and the time library which exposes the C library functions for manipulating dates and times.

The game screen, color, title, and size of the game screen are then defined using a turtle. Some variables have been initialized.

```
import turtle
import time

# define the turtle screen
display = turtle.Screen()
# set the background colour
display.bgcolor("black")
display.title("A DFS Maze Solving Program")
# set up the dimensions of the working window
display.setup(1300, 700)

# declare system variables
start_x = 0
start_y = 0
end_x = 0
end_y = 0
```

Five classes drawing images of turtles were used to build the maze:

- A blank is used to print the maze;
- The green one to show the cells visited;
- The blue turtle to show the border cells;
- The red turtle to represent the starting position;
- The yellow turtle to represent the end position and the solution path.

The skeleton of the maze with grid is defined.

```
# the five classes below are drawing turtle images to construct the maze.
# use white turtle to stamp out the maze
# define a Maze class
class Maze(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        # the turtle shape
        self.shape("square")
        # colour of the turtle
        self.color("white")
        # lift the pen so it do not leave a trail
        self.penup()
        # define the animation speed
        self.speed(0)


# use green turtles to show the visited cells
class Green(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("green")
        self.penup()
        self.speed(0)
```

```
# use blue turtle to show the frontier cells
class Blue(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("blue")
        self.penup()
        self.speed(0)


# use the red turtle to represent the start position
class Red(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("red")
        # point turtle to point down
        self.setheading(270)
        self.penup()
        self.speed(0)
```

```
# use the yellow turtle to represent the end position and the solution path
class Yellow(turtle.Turtle):  # code as above
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("yellow")
        self.penup()
        self.speed(0)
```

```
grid1 = [
    "+++++++++++++++++++++++++++++++++++++++++++++++++",
    "+ S             +                              +",
    "+ ++++++++++ +++++++++++++++ ++++++++ +++++ +",
    "+           +                 +        +     +",
    "++ +++++++ +++++++++++++++ ++++++++++++++++++++",
    "++ ++    + ++           + ++                  +",
    "++ ++ ++ + ++ ++ +++++ ++ ++ +++++++++++++++ +",
    "++ ++ ++ + ++ ++ +     ++ ++ ++ ++        ++ +",
    "++ ++ ++++ ++ ++++++++++ ++ ++ +++++ +++ ++ +",
    "++ ++   ++ ++             ++          +++ ++e+",
    "++ ++++ ++ ++++++++++++++++ +++++++++++++++ +",
    "++    + ++                   ++               +",
    "+++++ + ++++++++++++++++++++++ ++++++++++++ +",
    "++ ++ +                    ++          +++ ++ +",
    "++ ++ ++++ +++++++++++++++ ++ +++++ ++ +++ ++ +",
    "++ ++ ++   ++     +    ++ ++ ++    ++     ++ +",
    "++ ++ ++ +++++++ +++++ ++ ++ +++++++++++++++ +",
    "++                    ++ ++ ++               +",
    "+++++ ++ + ++++++++++ ++ ++ ++ +++++++++++++",
    "+++++++++++++++++++++++++++++++++++++++++++++++++",
]
```

This function builds the maze based on the type of grid defined above.

For each line in the grid, the characters defined in the various positions are assigned.

```
# this function constructs the maze based on the grid type above
def setup_maze(grid):
    # set up global variables for start and end locations
    global start_x, start_y, end_x, end_y
    # iterate through each line in the grid
    for y in range(len(grid)):
        # iterate through each character in the line
        for x in range(len(grid[y])):
            # assign the variable character to the y and x positions of the grid
            character = grid[y][x]
            # move to the x location on the screen staring at -288
            screen_x = -588 + (x * 24)
            # move to the y location of the screen starting at 288
            screen_y = 288 - (y * 24)

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

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

# if cell contains an 'e'
if character == "e":
    # move pen to the x and y location and
    yellow.goto(screen_x, screen_y)
    # stamp a copy of the yellow turtle on the screen
    yellow.stamp()
    # assign end locations variables to end_x and end_y
    end_x, end_y = screen_x, screen_y
    # add cell to the path list
    path.append((screen_x, screen_y))

# if cell contains a "s"
if character == "s":
    # assign start locations variables to start_x and start_y
    start_x, start_y = screen_x, screen_y
    # send red turtle to start position
    red.goto(screen_x, screen_y)
```

```
# check down
if (x, y - 24) in path and (x, y - 24) not in visited:
    cell_down = (x, y - 24)
    # backtracking routine [cell] is the previous cell. x, y is the current cell
    solution[cell_down] = x, y
    blue.goto(cell_down)
    blue.stamp()
    frontier.append(cell_down)

# check right
if (x + 24, y) in path and (x + 24, y) not in visited:
    cell_right = (x + 24, y)
    # backtracking routine [cell] is the previous cell. x, y is the current cell
    solution[cell_right] = x, y
    blue.goto(cell_right)
    blue.stamp()
    frontier.append(cell_right)
```

The function for the Depth First Search algorithm is defined. The ifs inside the function are defined to have the path in the maze that visits all possible roads. It defines the route in yellow after all possible roads have been visited and make sure that both the yellow and the red are still visible after they have been visited.

```
def searchDFS(x, y):
    # add the x and y position to the frontier list
    frontier.append((x, y))
    # add x and y to the solution dictionary
    solution[x, y] = x, y
    # loop until the frontier list is empty
    while len(frontier) > 0:
        # change this value to make the animation go slower
        time.sleep(0)
        # current cell equals x and  y positions
        current = (x, y)

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

```
# check up
if (x, y + 24) in path and (x, y + 24) not in visited:
    cell_up = (x, y + 24)
    # backtracking routine [cell] is the previous cell. x, y is the current cell
    solution[cell_up] = x, y
    blue.goto(cell_up)
    blue.stamp()
    frontier.append(cell_up)

# remove last entry from the frontier list and assign to x and y
x, y = frontier.pop()
# add current cell to visited list
visited.append(current)
# green turtle goto x and y position
green.goto(x, y)
# stamp a copy of the green turtle on the maze
green.stamp()
# makes sure the yellow end turtle is still visible after being visited
if (x, y) == (end_x, end_y):
    # stamp the yellow turtle at the end position
    yellow.stamp()
# makes sure the red start turtle is still visible after being visited
if (x, y) == (start_x, start_y):
    # stamp the red turtle at the start position
    red.stamp()
```

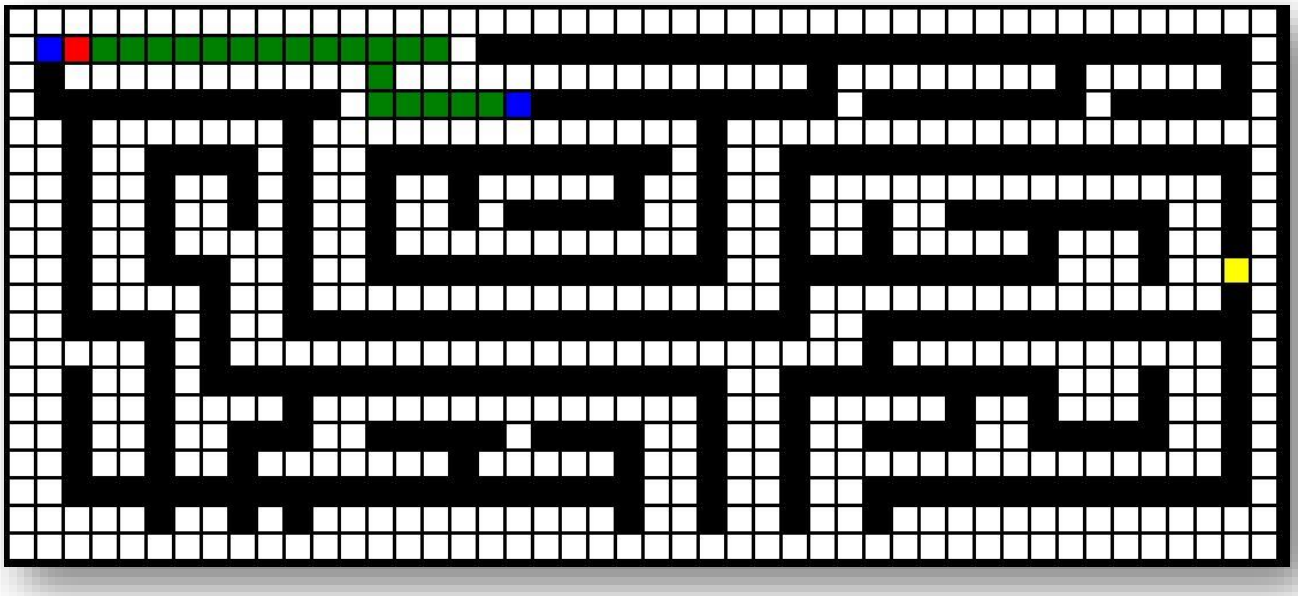
This function defines the final path. The loop stops when the current cells are equal to the starting cell.

```
# this is the solution path function
def backRoute(x, y):
    yellow.goto(x, y)
    yellow.stamp()
    # stop loop when current cells == start cell
    while (x, y) != (start_x, start_y):
        # move the yellow turtle to the key value of solution ()
        yellow.goto(solution[x, y])
        # create solution path
        yellow.stamp()
        # "key value" now becomes the new key
        x, y = solution[x, y]
```

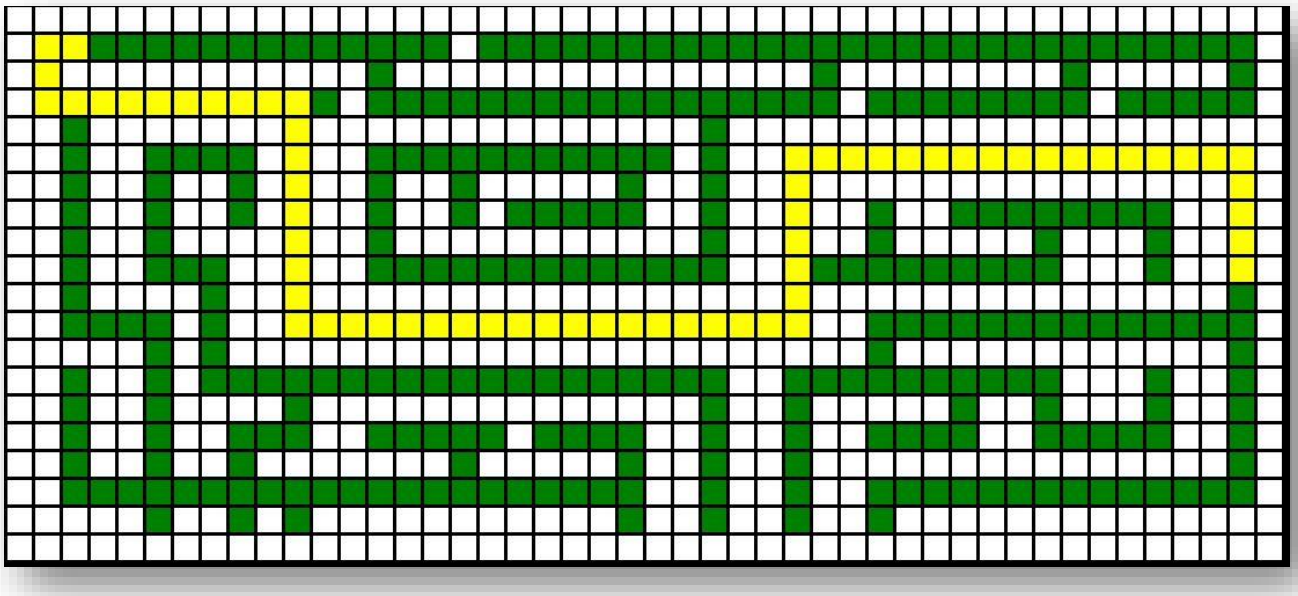
The empty lists that will be used are initialized and the defined functions are called.

```
#  initialise lists
maze = Maze()
red = Red()
blue = Blue()
green = Green()
yellow = Yellow()
walls = []
path = []
visited = []
frontier = []
solution = {}


if __name__ == '__main__':
    # call setup maze function
    setup_maze(grid)
    # call search function
    searchDFS(start_x, start_y)
    # call back route function
    backRoute(end_x, end_y)

# exit out Pygame when x is clicked
display.exitonclick()
```

Once the maze is created, the starting point is marked with a red square and the end point with a yellow square. We begin to explore all possible avenues to arrive at the solution.



Finally, the final path from the red dot to the yellow dot is marked in yellow.



NOTES: For change the start position and the end position, should be change the letter s and the letter e in the grid. Also the DFS and BFS files must be run separately (first one and then the other) as two different methods have been used: one uses only the turtle library and the other also uses sys and collections.

Report: Maze Game (Breadth First Search)

For this exercise, I used the Python turtle library which provides the primitives of the turtle graphics in both object-oriented and procedural form, and the time library which exposes the C library functions for manipulating dates and times. The sys module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter. It is always available. The collections module implements specialized container data types by providing alternatives to Python's built-in containers, dict, list, set, and tuples for general use.

The game screen, color, title, and size of the game screen are then defined using a turtle.

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

# define the turtle screen
wn = turtle.Screen()
# set the background colour
wn.bgcolor("black")
wn.title("A BFS Maze Solving Program")
# set up the dimensions of the working window
wn.setup(1300, 700)
```

Five classes drawing images of turtles were used to build the maze:

- A blank is used to print the maze;
- The green one to show the cells visited;
- The blue turtle to show the border cells;
- The red turtle to represent the starting position;
- The yellow turtle to represent the end position and the solution path.

The skeleton of the maze with grid is defined.

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

```
class Blue(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("square")
        self.color("red")
        self.penup()
        self.speed(0)


class Yellow(turtle.Turtle):
    def __init__(self):
        turtle.Turtle.__init__(self)
        self.shape("square")
        self.color("yellow")
        self.penup()
        self.speed(0)
```

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

This function builds the maze based on the type of grid defined above.

For each line in the grid, the characters defined in the various positions are assigned.

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

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

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

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

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

This function define the end of the program.

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

The function for the Breadth First Search algorithm is defined. The ifs inside the function are defined to have the path in the maze that visits all possible roads. It defines the route in yellow after all possible roads have been visited and make sure that both the yellow and the red are still visible after they have been visited.

```
def searchBFS(x, y):
    frontier.append((x, y))
    solution[x, y] = x, y

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

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

```
# check the cell down
if (x, y - 24) in path and (x, y - 24) not in visited:
    cell = (x, y - 24)
    solution[cell] = x, y
    frontier.append(cell)
    visited.add((x, y - 24))
    print(solution)

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

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

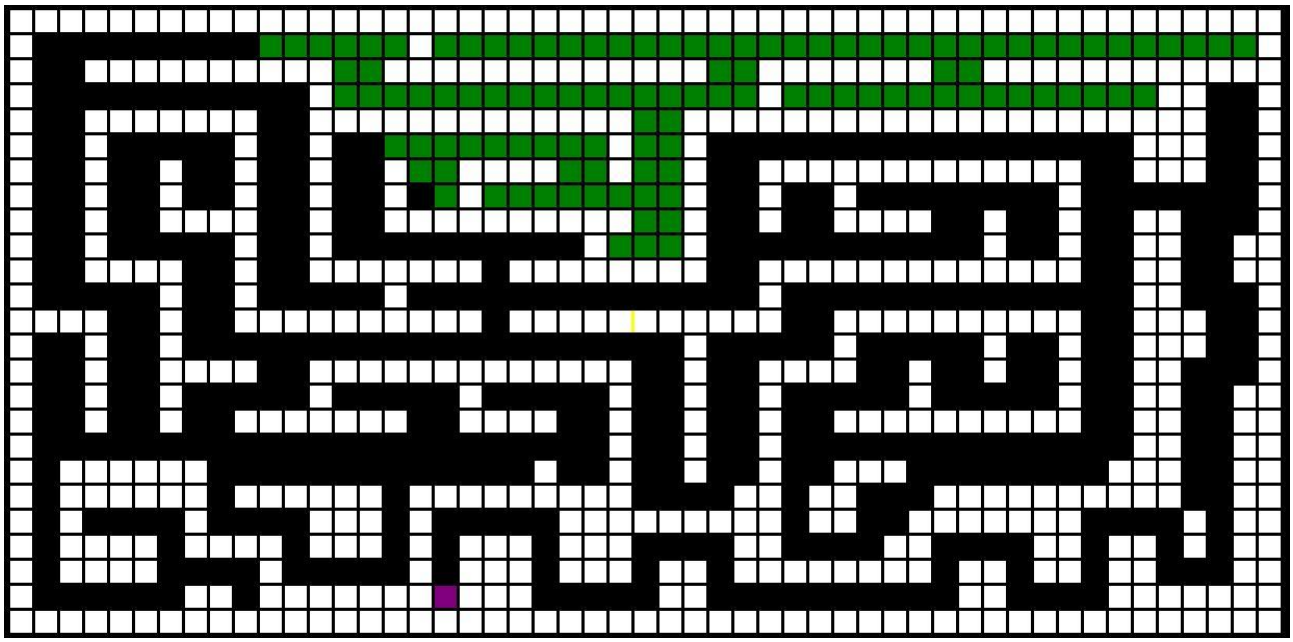
This function defines the final path. The loop stops when the current cells are equal to the starting cell.

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

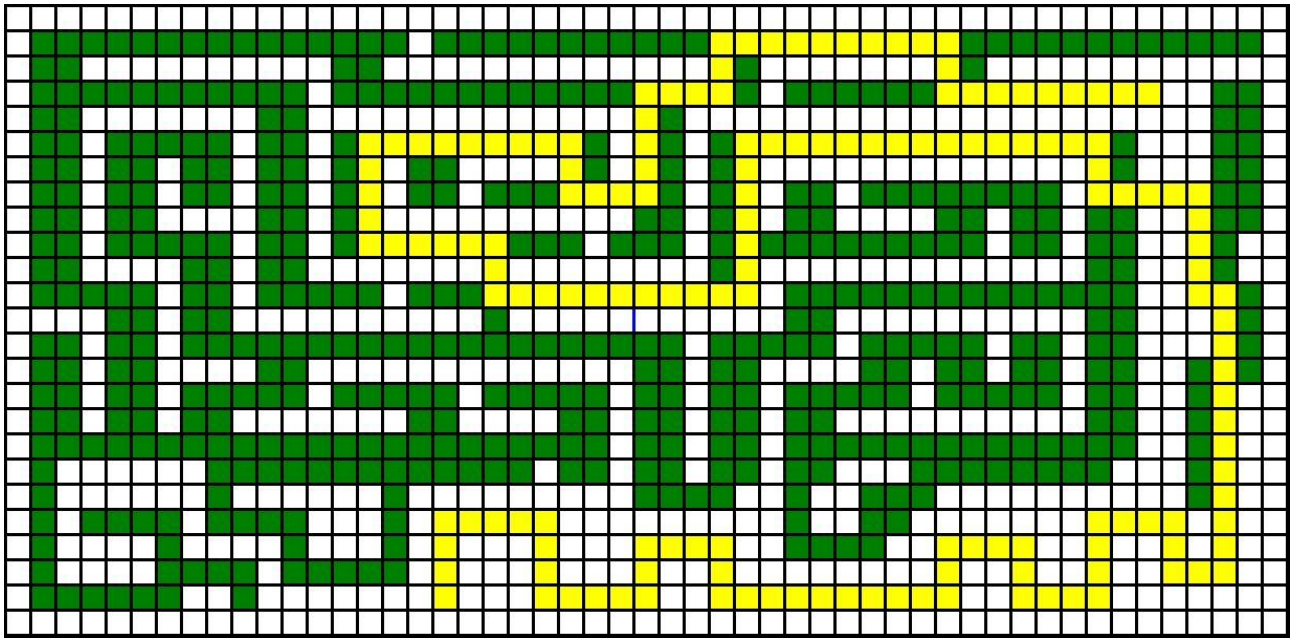
The empty lists that will be used are initialized and the defined functions are called.

```
maze = Maze()
red = Red()
blue = Blue()
green = Green()
yellow = Yellow()

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

# declare system variables
start_x = 0
start_y = 0
end_x = 0
end_y = 0

if __name__ == '__main__':
    # main program starts here ####
    setup_maze(grid)
    searchBFS(start_x, start_y)
    backRoute(end_x, end_y)
    wn.exitonclick()
```

Once the maze is created, the starting point is marked with a red square and the end point with a purple square. We begin to explore all possible avenues to arrive at the solution.



Finally, the final path from the red dot to the purple dot is marked in yellow.



NOTES: For change the start position and the end position, should be change the letter s and the letter e in the grid. Also the DFS and BFS files must be run separately (first one and then the other) as two different methods have been used: one uses only the turtle library and the other also uses sys and collections.

Statistics DFS:

The time that the algorithm takes to find the solution since the program is run, the labyrinth is built, all the possible ways have been found and the final solution has been found is 00:24.54.

The time that the algorithm takes to find the solution with the maze already built and therefore it only has to find all the possible roads and the final solution is 00:14.00.

Statistics BFS:

The time that the algorithm takes to find the solution since the program is run, the labyrinth is built, all the possible ways have been found and the final solution has been found is 00:32.59.

The time that the algorithm takes to find the solution with the maze already built and therefore it only has to find all the possible roads and the final solution is 00:20.94.