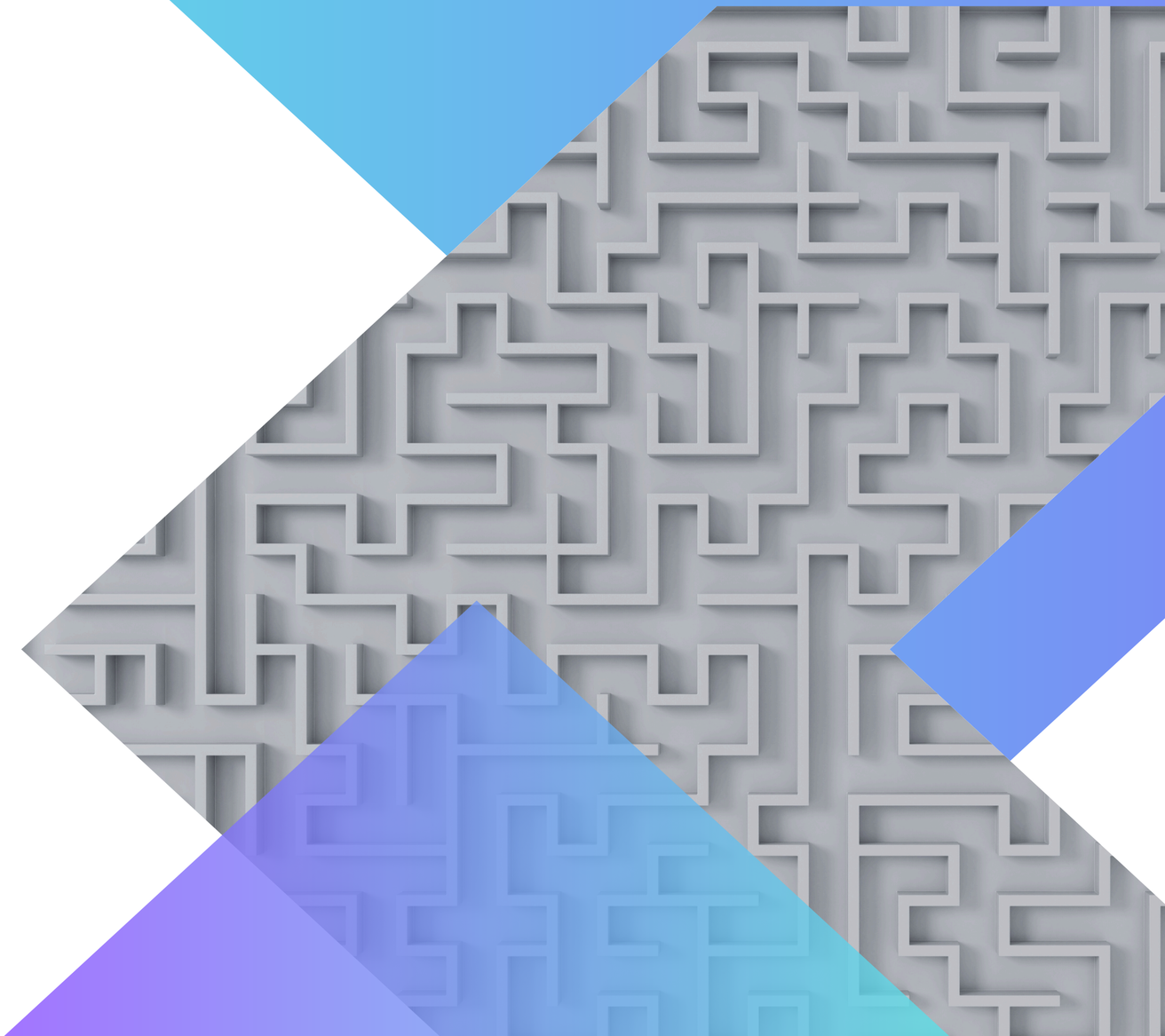
ZAGAZIG UNIVERSITY

Maze Solver

Computer and Systems Engineering



Under Supervision

Dr/Al-Shimaa Nabil

Our Team

	Name	Code
1	Reham Hamdi Mohamed Ibrahim	20812021201109
2	Shahd Elsayed Ahmed Ahmed	20812021200543
3	Matilda Ashraf Malak Mikhail	20812021201241
4	Mariam Farouk Slama Adly	20812021201289
5	Nancy Ayman Nabil Mohamed	20812021200506

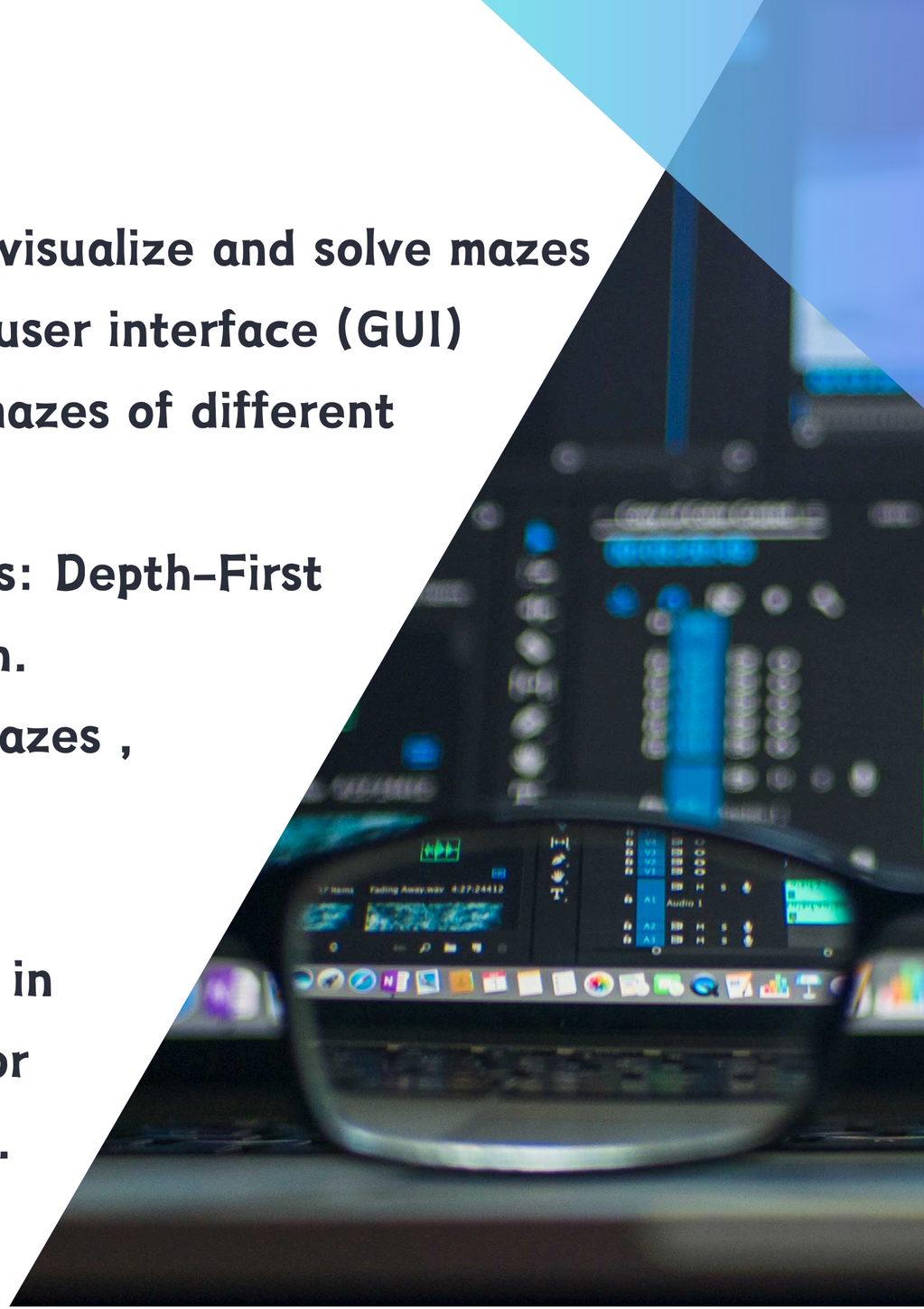
Abstract

The maze solver system is a Java application designed to visualize and solve mazes using various algorithms.The system provides a graphical user interface (GUI) where users can interactively create, modify, and solve mazes of different sizes and complexities.

It offers implementations of three maze-solving algorithms: Depth-First search (DFS), Breadth-First Search (BFS), and A* Search.

Users can open pre-definec maze files , draw their own mazes , select the start, and end points, and apply algorithms to find the shortest path between them.

The system renders the maze grid and algorithm progress in real-time, providing a dynamic and intuitive experience for users to understand and explore maze-solving techniques.



Maze solver system components

- **Main GUI:**

The main graphical user interface where users interact with the system. It includes options to open maze files, draw mazes. select start and end points, apply algorithms, and clear search results

- **Node Class:**

Represents individual cells or nodes in the maze grid. Each node contains information such as its position, color, and neighboring nodes. It provides methods for rendering and manipulation

- **Algorithm Class:**

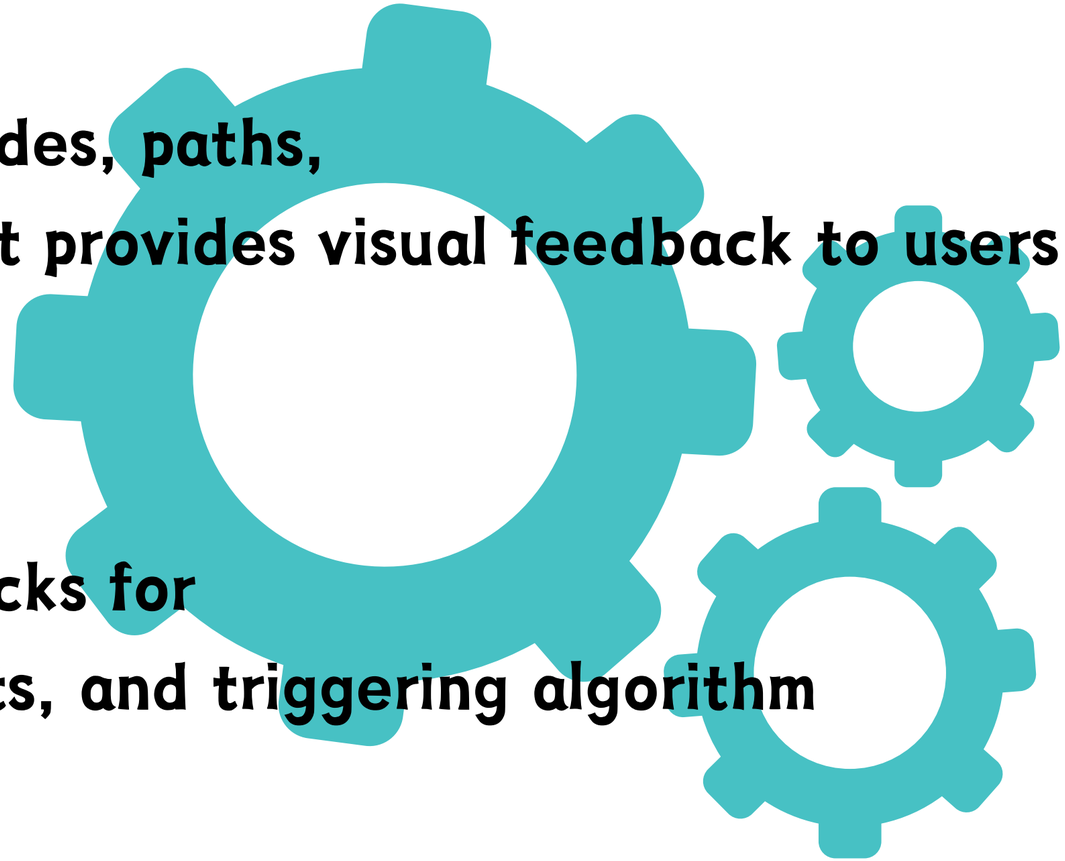
Implements various maze-solving algorithms, including Depth-First Search (DFS), Breadth-First Search (BFS), and A* Search. These algorithms traverse the maze grid, marking nodes as searched and finding the shortest path between the start and end points.

- **File 1/O:**

Handles reading maze files from external sources. It allows users to open pre-defined maze configurations and load them into the system for solving

- **Graphics Rendering:**

Responsible for rendering the maze grid, nodes, paths, and algorithm progress on the GUI canvas. It provides visual feedback to users as algorithms search and solve the maze.



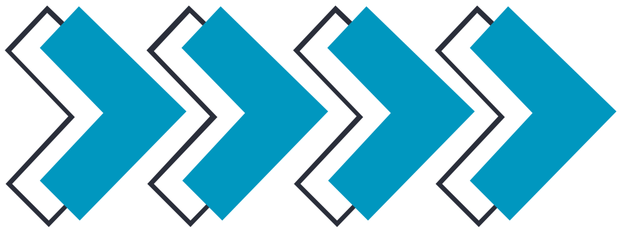
- **User Input Handling:**

Captures user interactions such as mouse clicks for drawing mazes, selecting start and end points, and triggering algorithm execution.

Maze Solver System Real-Life Applications

The maze solver system plays a crucial role in various real-life applications where efficient pathfinding and navigation complex environments are required. Its ability to find optimal paths through mazes can improve efficiency, safety and decision-making in diverse fields ranging from robotics and video games to transportation and emergency management.

REAL LIFE APPLICATION EXAMPLES:



1. Routing and Pathfinding in Transportation
2. Network Routing and Optimization
3. Emergency Evacuation Planning
4. Artificial Intelligence/Robotics
5. Computer Games

Maze Solver System Suitable Algorithms:

Graph searching algorithms are well-suited for maze solver systems due to their ability to represent mazes as graphs, systematically traverse maze paths, handle various mazestructures, ensure optimality and efficiency, and adapt to different maze characteristics.

Graph Searching Algorithms:

Searching a graph:

- Systematically follow the edges of a graph to visit the vertices of the graph

Used to discover the structure of a graph.

The Algorithms used in our project:

- Breadth-first Search (BFS)
- Depth-first Search (DFS).
- A star (A*).

Breadth First Search Algorithm

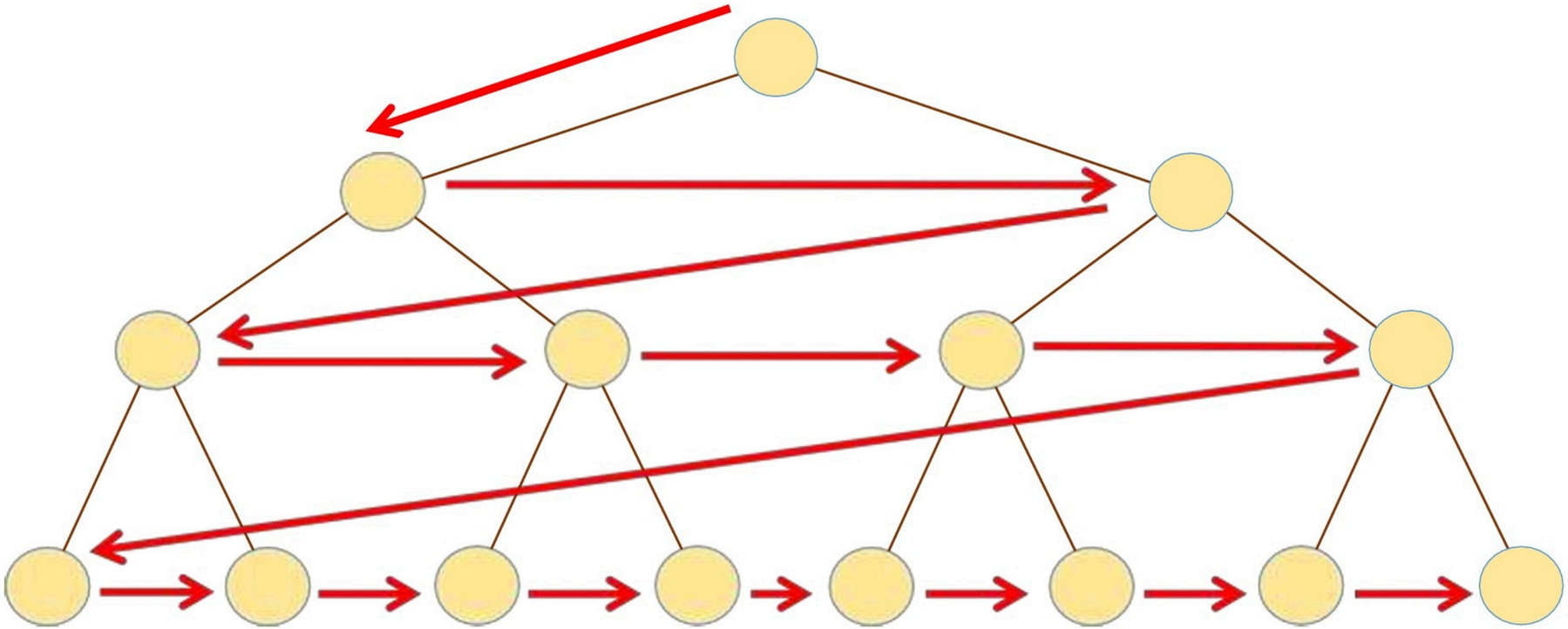
Breadth First Search (BSF):

- BFS is an example of a brute-force & Exhaustive search algorithm.

- Initially all vertices of the graph are unvisited
- Start visiting the graph from any vertex, say v
  - Visit each unvisited adjacent vertex of v.
  - Repeat the process for each vertex visited.
- This process stops when alll vertices reachable from v are visited.

Breadth-First Search (BFS)

Level by level



Bredth First Search Rules:

- Rule 0:Insert initial node in the queue.
- Rule 1:Remove the head of the queue and Mark it as visited.
- Rule 2:Insert all adjacent nodes (to removed one) into queue
- Rule 3:If no adjacent vertex is found, then stop..
- Rule 4:Repeat Rule 1 to Rule 3 until the queue is empty.

pseudo-code

BFS

```
Set all nodes to "not visited";                          }O(V)

q = new Queue();                                         }O(1)

q.enqueue(initial node);                                 }O(1)

while ( q ≠ empty ) do
{
  x = q.dequeue();                                       }O(1)

  if ( x has not been visited )                          }O(1)
  {
    visited[x] = true;          // Visit node x !        }O(1)

    for ( every edge (x, y)  /* we are using all edges ! * }O(E)
      if ( y has not been visited )
            q.enqueue(y);       // Use the edge (x,y) !!  }O(1)
  }
}
```

O(E)

Total Time Complexity

O(V + E)

Performance of BFS Algorithm

- **Best Case: O(1)**
  - This typically occurs when the initial node is the goal node, so you don't need to search further. In such a scenario, BFS would only need to examine the initial node, resulting in constant time complexity.
- **Average Case: O(V + E)**
  - In an average case, where V represents the number of vertices (nodes) and E represents the number of edges in the graph, BFS visits each vertex and each edge once. Hence, the time complexity is linear in terms of the number of vertices and edges.
- **Worst Case: O(V + E)**
  - Similar to the average case, BFS traverses each vertex and edge once. Thus, the worst-case time complexity is also linear with respect to the number of vertices and edges.

- **Advantages:**
  - BFS guarantees finding the shortest path to the goal. It explores nodes level by level, ensuring that shorter paths are found before longer ones.
  - It is complete and optimal, meaning it will always find the shortest path if one exists.

- **Disadvantages:**
  - BFS typically requires more memory than DFS because it needs to store information about all nodes at each level of the search tree.
  - It may be slower than DFS for large mazes or mazes with many possible paths because it explores all neighbors of a node before moving to the next level.

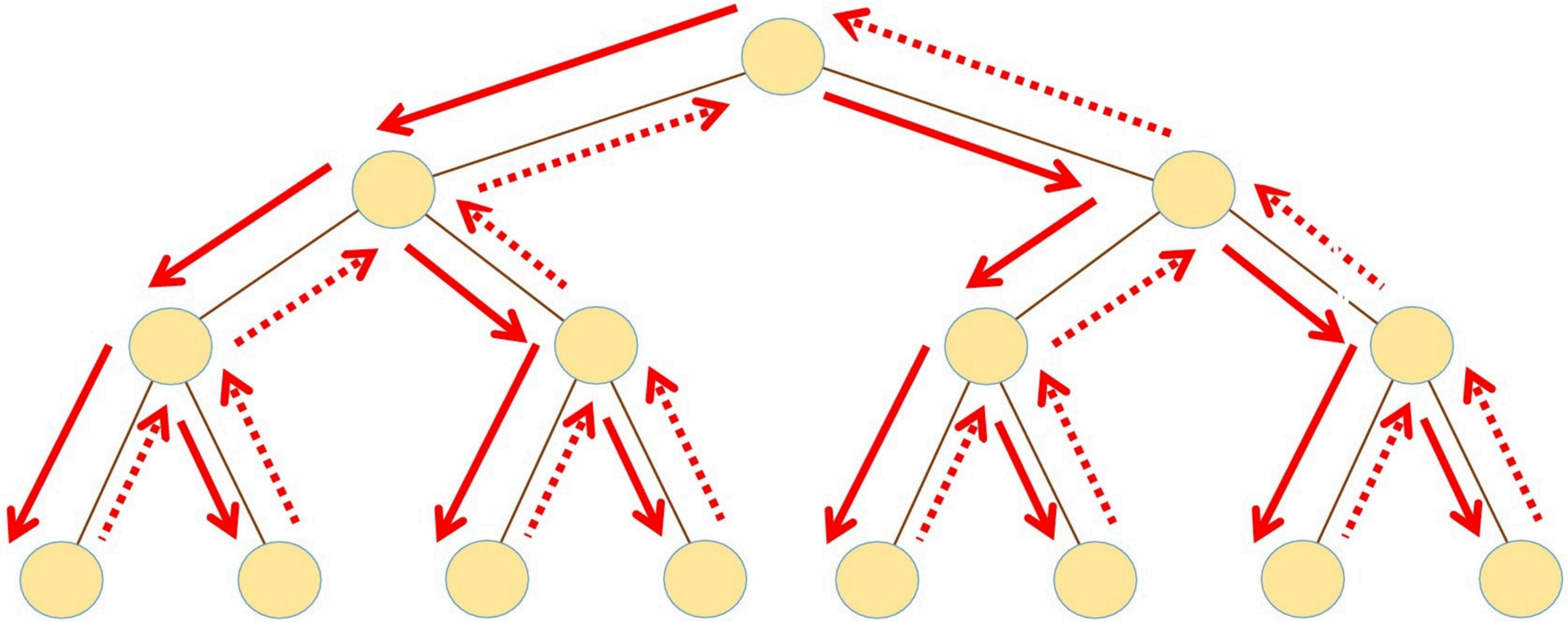
Depth First Search Algorithm

Depth-first Search (DFS):

- DFS is an example of a brute-force & Exhaustive search algorithm.
- Explore edges out of the most recently discovered vertex v.
- When all edges of v have been explored, backtrack to explore other edges leaving the vertex from which v was discovered
- "Search as deep as possible first."
- Continue until all vertices reachable from the original source are discovered.

Depth-First Search (DFS)

Depth by depth



Depth First Search Steps:

1. Visit the start node (or current node) and push onto a stack so we can remember it and then mark it so we won't visit it again

2. Go to any node adjacent to the current node that has not been visited yet

3. it as visited, and push it onto the stack

4. Continue visiting the next adjacent nodes, until you reach a node that has no other adjacent nodes.

5. Next pop off nodes from the stack until you find a node that has other unvisited nodes adjacent to it that you can evaluate.

6. Repeat steps 2 through 5 until end is reached.

pseudo-code

DFS

```
DFS(G,v)( v is the vertex where the search starts )
        Stack S := {};    ( start with an empty stack )
        for each vertex u, set visited[u] := false;
        push S, v;
        while (S is not empty) do
            u := pop S;
            if (not visited[u]) then
                visited[u] := true;
                for each unvisited neighbour w of u
                    push S, w;
            end if
        end while
    END DFS()
```

}O(1)
}O(1)
}O(V)
}O(1)
}O(1)
}O(1)
}O(1)
}O(E)
}O(1)

O(E)

Total Time Complexity

O(V + E)

Performance of DFS Algorithm

- **Best Case: O(1)**
  - As with BFS, the best case occurs when the initial node is the goal node. In this scenario, DFS would only need to examine the initial node, resulting in constant time complexity.
- **Average Case: O(V + E)**
  - DFS traverses each vertex and each edge once, like BFS. Therefore, the average-case time complexity is linear in terms of the number of vertices and edges.
- **Worst Case: O(V + E)**
  - Similar to BFS, DFS traverses each vertex and edge once. Hence, the worstcase time complexity is also linear with respect to the number of vertices and edges.

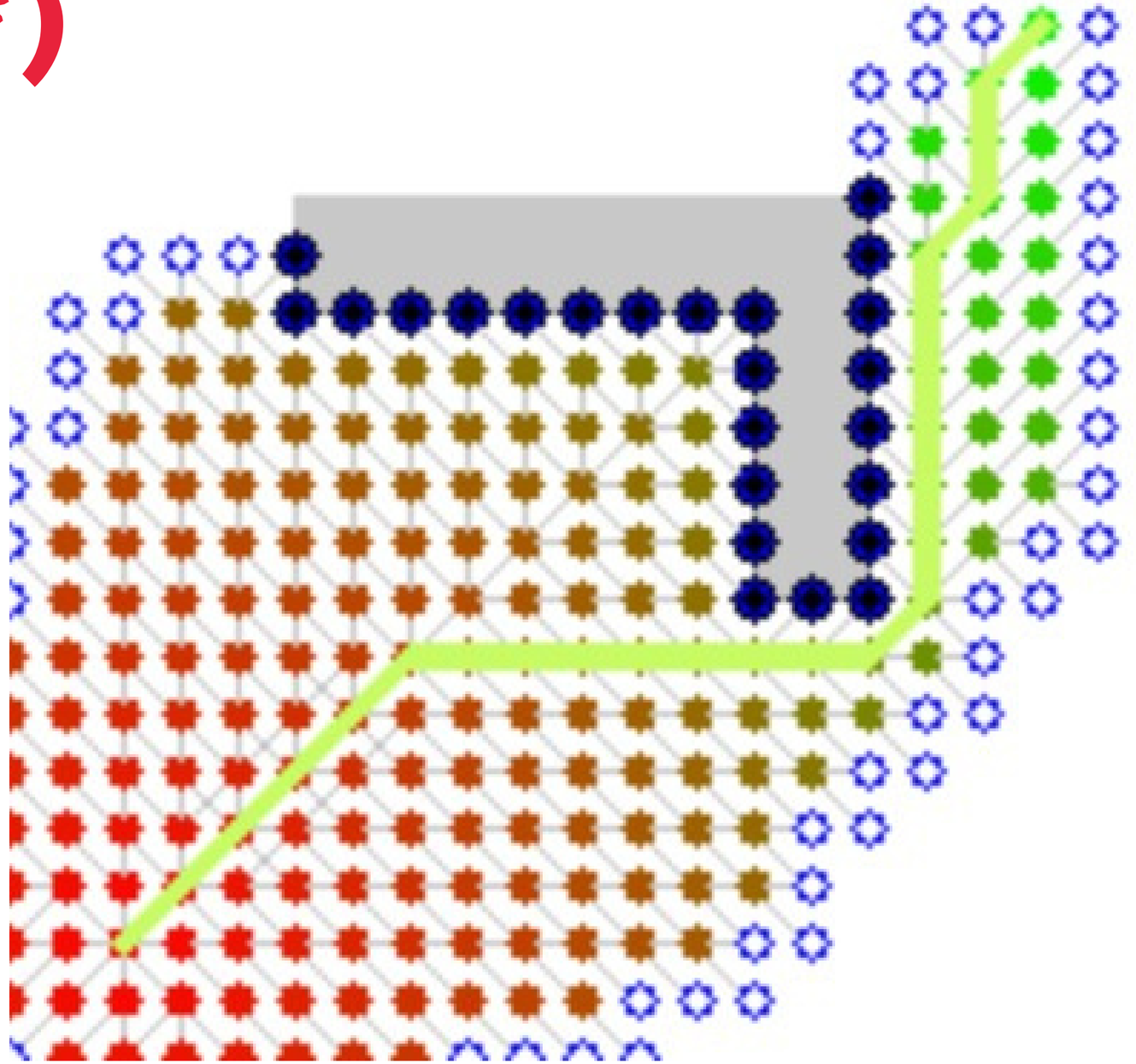
- **Advantages:**
  - DFS is straightforward to implement and requires minimal memory overhead, typically utilizing a stack to keep track of the path.
  - It can be particularly efficient in situations where there are many paths to explore, as it goes deep into one path before backtracking.
- **Disadvantages:**
  - DFS does not guarantee finding the shortest path. It may traverse deep into one branch of the maze before exploring other branches, potentially missing shorter paths.
  - In some cases, DFS may get stuck in infinite loops if the maze contains cycles.
  - DFS may be inefficient for large mazes or mazes with many possible paths, as it explores one path to its end before considering others.

A Star Algorithm

A Star (A*):

- A* search algorithm is a path finding algorithm that finds the single-pair shortest path between the start node(source) and the target node(destination) of a weighted graph.
- The algorithm not only considers the actual cost from the start node to the current node(g) but also tries to estimate the cost will take from the current node to the target node using heuristics (h)
- Then it selects the node that has the lowest f-value(f=g+h) to be the next node to move until it hits the target node .

A Star (A*)



A Star Steps:

- Input: A* is a graph search algorithm , which take a “graph” as input. A graph is a set of locations (“nodes”) and the connections (“edges”) between them.

- A* selects the path that minimizes f(n)=g(n)+h(n) where n is the next node on the path, g(n) is the cost of the path from the start node to n, and h(n) is a heuristic function that estimates the cost of the . cheapest path from n to the goal.

pseudo-code

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = {}
cost_so_far = {}
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
   current = frontier.get()

   if current == goal:
      break

   for next in graph.neighbors(current):
      new_cost = cost_so_far[current] + graph.cost(current, next)
      if next not in cost_so_far or new_cost < cost_so_far[next]:
         cost_so_far[next] = new_cost
         priority = new_cost + heuristic(goal, next)
         frontier.put(next, priority)
         came_from[next] = current
```

O(1)
O(log V)
O(1)
O(1)
O(1)
O(log V)
O(1)
O(E)
O(1)
O(1)
O(1)
O(log V)
O(1)

O(E logV)

Total Time Complexity

O(E logV)

Performance of A* Algorithm

- **Best Case: O(logV)**
  - In the best-case scenario, where we have a very efficient heuristic function guiding us directly to the goal, each node is expanded only once, and the priority queue operations take O(logV) time.
- **Average Case: O(ElogV).**
  - In the average case, the A* algorithm explores nodes based on the heuristic information provided. The number of nodes expanded could vary, but typically it's not exploring all possible paths.
  - The time complexity of each priority queue operation is still O(logV), but this time we perform this operation for every edge we encounter, leading to O(ElogV) time complexity.

- **Worst Case: O(ElogV)**
  - In the worst-case scenario, where the heuristic function doesn't provide much guidance and the algorithm needs to explore a large portion of the graph, each node can be expanded multiple times.
  - Again, for each expansion, the priority queue operations take O(logV) time. However, this time we perform these operations for every edge encountered, leading to O(ElogV) time complexity.
- **Advantages:**
  - A* is informed and uses a heuristic function to guide the search towards the goal efficiently.
  - It is complete and optimal when using an admissible heuristic, meaning it will always find the shortest path if one exists.
  - A* often outperforms both DFS and BFS, especially for larger mazes or mazes with complex structures, as it focuses on promising paths based on the heuristic estimate.

- **Disadvantages:**
  - The performance of A* heavily depends on the quality of the heuristic function. If the heuristic is not admissible or overly optimistic, A* may not find the optimal solution.
  - A* may require more memory and computational resources compared to DFS and BFS due to the need to maintain priority queues and evaluate heuristic functions.

Comparison of Searching Algorithms

Criteria	BFS	DFS	A*
Technique	Brute-force & Exhaustive search	Brute-force & Exhaustive search	Greedy heuristic algorithm
Worst Case	O(V + E)	O(V + E)	O(E logV)
Average Case	O(V + E)	O(V + E)	O(E logV)
Best Case	O(1)	O(1)	O(log V)
Space Complexity	O(V)	O(V)	O(V)

Code Analysis

```java
public class Node {
    private int Xpos;
    private int Ypos;
    private Color nodeColor = Color.LIGHT_GRAY;
    private final int WIDTH = 35;
    private final int HEIGHT = 35;
    private Node left, right, up, down; // Represent references to neighboring nodes.
    public Node(int x, int y) {
        Xpos = x;
        Ypos = y;
    }
    public Node() {
    }//Computes the Euclidean distance between two nodes a and b.
    public static double distance(Node a, Node b) {
        if (a == null || b == null) {
            // Handle null case (e.g., return Double.POSITIVE_INFINITY)
            return Double.POSITIVE_INFINITY;
        }
        double x = Math.pow(a.Xpos - b.Xpos, 2);
        double y = Math.pow(a.Ypos - b.Ypos, 2);
        return Math.sqrt(x + y);
    }//Renders the node on the graphics context g.
    //Draws a black border and fills the node with its assigned color.
    public void render(Graphics2D g) {
        g.setColor(Color.BLACK);
        g.drawRect(Xpos, Ypos, WIDTH, HEIGHT);
        g.setColor(nodeColor);
        g.fillRect(Xpos + 1, Ypos + 1, WIDTH - 1, HEIGHT - 1);
    }//Changes the color of the node based on the mouse button clicked.
    public void Clicked(int buttonCode) {
        if (buttonCode == 1) {
            // WALL
            nodeColor = Color.BLACK;
        }
        if (buttonCode == 2) {
            // START
            nodeColor = Color.GREEN;
        }
        if (buttonCode == 3) {
            // END
            nodeColor = Color.RED;
        }
        if (buttonCode == 4) {
            // CLEAR
            clearNode();
        }
    }
    public void setColor(Color c) {
        nodeColor = c;}
    public Color getColor() {
        return nodeColor;
    }//Returns a list of neighboring nodes that are traversable.
    public List<Node> getNeighbours() {
        List<Node> neighbours = new ArrayList<>();
        if (left != null && left.isPath()) {
            neighbours.add(left);}
        if (down != null && down.isPath()) {
            neighbours.add(down);}
        if (right != null && right.isPath()) {
            neighbours.add(right);}
        if (up != null && up.isPath()) {
            neighbours.add(up);}
        return neighbours;
    }
    public void setDirections(Node l, Node r, Node u, Node d) {
        left = l;
        right = r;
        up = u;
        down = d;
    }
    public void clearNode() {
        nodeColor = Color.LIGHT_GRAY;}
    public int getX() {
        return (Xpos - 15) / WIDTH;}
    public int getY() {
        return (Ypos - 15) / HEIGHT;}
    public Node setX(int x) {
        Xpos = x;
        return this;}
    public Node setY(int y) {
        Ypos = y;
        return this;}
    public boolean isWall() {
        return (nodeColor == Color.BLACK);}
    public boolean isStart() {
        return (nodeColor == Color.GREEN);}
    public boolean isEnd() {
        return (nodeColor == Color.RED);}
    public boolean isPath() {
        return (nodeColor == Color.LIGHT_GRAY || nodeColor == Color.RED);
    }//blue nodes means node that has been searched in certain algorithms.
    public boolean isSearched() {
        return (nodeColor == Color.BLUE || nodeColor == Color.ORANGE);
    }
}
```

O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

O(1)

```java
public class Algorithm {
    long startTime;// start time
    long stopTime;//stop time
    long duration;//calculate the elapsed time
    double dfsTime; //time for DFS
    double bfsTime; //time for BFS
    double AstarTime; //time for A*
    int nodeCount = 0; // Counter for the number of nodes visited
    public void dfs(Node start) {
        startTime = System.nanoTime(); // start of the time
        Stack<Node> nodes = new Stack<>();
        nodes.push(start);  nodeCount = 0;
        while (!nodes.empty()) {
            Node curNode = nodes.pop();
            if (!curNode.isEnd()) {
                curNode.setColor(Color.ORANGE);
                curNode.setColor(Color.BLUE); //visited node
                nodeCount++; // Increment the node count
                for (Node adjacent : curNode.getNeighbours()) {
                    nodes.push(adjacent);  }
            } else { curNode.setColor(Color.MAGENTA); //end node
                break;}  }
         if (!nodes.isEmpty()) {//you exited before the satck was emptied, meaning that y
            stopTime = System.nanoTime(); }   //stop the timer
        duration = stopTime - startTime;    //calculate the elapsed time
        dfsTime = (double) duration / 1000000;   //convert to ms
        System.out.println(String.format("DFS Algorithm Time %1.3f ms", dfsTime));
        System.out.println("Number of nodes traversed in DFS Algorithm: " + nodeCount); }
    public void bfs(Node start, Node end, int graphWidth, int graphHeight) {  // start of
        startTime = System.nanoTime();
        Queue<Node> queue = new LinkedList<>();
        Node[][] prev = new Node[graphWidth][graphHeight];
        queue.add(start);  nodeCount = 0;
        while (!queue.isEmpty()) {
            Node curNode = queue.poll();
            if (curNode.isEnd()) {
                curNode.setColor(Color.MAGENTA);
                break;  }
            if (!curNode.isSearched()) {
                curNode.setColor(Color.ORANGE);
                curNode.setColor(Color.BLUE);
                nodeCount++; // Increment the node count
                for (Node adjacent : curNode.getNeighbours()) {
                    queue.add(adjacent);
                    prev[adjacent.getX()][adjacent.getY()] = curNode; } } }
        if (!queue.isEmpty()) {  //you exited before the queue was emptied, meaning that y
            stopTime = System.nanoTime(); }   //stop the timer
        shortpath(prev, end);
        duration = stopTime - startTime;    //calculate the elapsed time
        bfsTime = (double) duration / 1000000;   //convert to ms
        System.out.println(String.format("BFS Algorithm Time %1.3f ms", bfsTime));
        System.out.println("Number of nodes traversed in BFS Algorithm: " + nodeCount); }
    public void Astar(Node start, Node targetNode, int graphWidth, int graphHeight) {
        startTime = System.nanoTime();
        PriorityQueue<Node> queue = new PriorityQueue<>((n1, n2) -> {
            double f1 = Node.distance(n1, targetNode) + Node.distance(start, n1);
            double f2 = Node.distance(n2, targetNode) + Node.distance(start, n2);
            return Double.compare(f1, f2);  });
        Node[][] prev = new Node[graphWidth][graphHeight];
        queue.add(start);   nodeCount = 0;
        while (!queue.isEmpty()) {
            Node curNode = queue.poll();
            if (curNode.isEnd()) {
                curNode.setColor(Color.MAGENTA);
                break;  }
            curNode.setColor(Color.ORANGE);
            curNode.setColor(Color.BLUE);  nodeCount++;
            for (Node adjacent : curNode.getNeighbours()) {
                if (adjacent.isSearched()) { continue;  }
                double f1 = Node.distance(adjacent, targetNode);
                double h1 = Node.distance(curNode, start);
                double f2 = Node.distance(adjacent, targetNode);
                double h2 = Node.distance(prev[adjacent.getX()][adjacent.getY()], start);
                if (!queue.contains(adjacent) || (f1 + h1 < f2 + h2)) {
                    prev[adjacent.getX()][adjacent.getY()] = curNode;
                    if (!queue.contains(adjacent)) {
                        queue.add(adjacent); }  }  }  }
        shortpath(prev, targetNode);
        if (!queue.isEmpty()) {   //you exited before the queue was emptied, meaning that y
            stopTime = System.nanoTime();  }
        duration = stopTime - startTime;
        AstarTime = (double) duration / 1000000;
        System.out.println(String.format("A* Algorithm Time %1.3f ms", AstarTime));
        System.out.println("Number of nodes traversed in A* Algorithm: " + nodeCount); }
    private void shortpath(Node[][] prev, Node end) {
        Node pathConstructor = end;
        while (pathConstructor != null) {
            pathConstructor = prev[pathConstructor.getX()][pathConstructor.getY()];
            if (pathConstructor != null) {
                pathConstructor.setColor(Color.ORANGE);}}}}
```

O(1)

O(V + E)

O(V + E)

O(E logV)

O(m * n)

```java
public class Main extends Canvas implements Runnable, MouseListener {
private static Node start = null;
    private static Node target = null;
    private static JFrame frame;
    private Node[][] nodeList;
    private static Main runTimeMain;
    private static Algorithm algorithm;
    private final static int WIDTH = 1000;
    private final static int HEIGHT = 750;
    private final static int NODES_WIDTH = 28;
    private final static int NODES_HEIGHT = 19;
    public static void main(String[] args) {
        frame = new JFrame("Maze Solver");
        frame.setSize(WIDTH, HEIGHT);
        frame.setResizable(false);
        frame.setLayout(null);
        Main m = new Main();
        algorithm = new Algorithm();
        m.setBounds(0, 25, WIDTH, HEIGHT);
        SetupMenu(frame);
        runTimeMain = m;
        frame.add(m);
        frame.setVisible(true);
        m.start();
    }
    public static void SetupMenu(JFrame frame) {
        JMenuBar bar = new JMenuBar();
        bar.setBounds(0, 0, WIDTH, 25);
        frame.add(bar);
        JMenu fileMenu = new JMenu("File");
        bar.add(fileMenu);
        JMenu boardMenu = new JMenu("Board");
        bar.add(boardMenu);
        JMenu algorithmsMenu = new JMenu("Algorithms");
        bar.add(algorithmsMenu);
        JMenuItem openMaze = new JMenuItem("Open Maze");
        JMenuItem clearSearch = new JMenuItem("Clear Search Results");
        JMenuItem bfsItem = new JMenuItem("Breadth-First Search");
        JMenuItem dfsItem = new JMenuItem("Depth-First Search");
        JMenuItem AstarItem = new JMenuItem("A-star Search");
        openMaze.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                try {
                    runTimeMain.openMaze();
                } catch (IOException e) { // TODO Auto-generated catch block
                    e.printStackTrace();}
            }
        clearSearch.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                runTimeMain.clearSearchResults();
            }
        });
        bfsItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                algorithm.bfs(runTimeMain.start, runTimeMain.target, runTimeMain.NODES_WIDTH,
                        runTimeMain.NODES_HEIGHT);
            }
        });
        dfsItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                algorithm.dfs(runTimeMain.getStart());
            }
        });
        AstarItem.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent arg0) {
                algorithm.Astar(runTimeMain.start, runTimeMain.target, runTimeMain.NODES_WIDTH,
                        runTimeMain.NODES_HEIGHT);
            }
        });
        fileMenu.add(openMaze);
        boardMenu.add(clearSearch);
        algorithmsMenu.add(dfsItem);
        algorithmsMenu.add(bfsItem);
        algorithmsMenu.add(AstarItem);
    }//Initializes the program and enters the game loop.
    public void run() {
        init();
        while (true) {
            BufferStrategy bs = getBufferStrategy(); // check
            if (bs == null) {
                createBufferStrategy(2);
                continue;
            }
            Graphics2D grap = (Graphics2D) bs.getDrawGraphics(); // check
            render(grap);
            bs.show();
            try {
                Thread.sleep(1);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }//Initializes the program by setting focus to the canvas,
```

O(1)

O(1)

O(1)

```java
//adding a mouse listener, creating a grid of nodes, and setting directions for each noc
    public void init() {
        requestFocus();
        addMouseListener(this);
        nodeList = new Node[NODES_WIDTH][NODES_HEIGHT];
        createNodes(false);
        setMazeDirections();
    }
    //Sets directions for each node in the grid based on its neighboring nodes.
    public void setMazeDirections() {
        for (int i = 0; i < nodeList.length; i++) {
            for (int j = 0; j < nodeList[i].length; j++) {
                Node up = null, down = null, left = null, right = null;
                int u = j - 1;
                int d = j + 1;
                int l = i - 1;
                int r = i + 1;
                if (u >= 0) {
                    up = nodeList[i][u];
                }
                if (d < NODES_HEIGHT) {
                    down = nodeList[i][d];
                }
                if (l >= 0) {
                    left = nodeList[l][j];
                }
                if (r < NODES_WIDTH) {
                    right = nodeList[r][j];
                }

                nodeList[i][j].setDirections(left, right, up, down);
            }
        }
    }
    public void createNodes(boolean ref) {
        for (int i = 0; i < nodeList.length; i++) {
            for (int j = 0; j < nodeList[i].length; j++) {
                if (!ref) {
                    nodeList[i][j] = new Node(i, j).setX(15 + i * 35).setY(15 + j * 35);
                }
                nodeList[i][j].clearNode();
            }
        }
    }
```

O(m * n)

O(m * n)

O(m * n)

```java
    public void openMaze() throws IOException {
        JFileChooser fileChooser = new JFileChooser();
        int option = fileChooser.showOpenDialog(frame);
        if (option == JFileChooser.APPROVE_OPTION) {
            File file = fileChooser.getSelectedFile();
            BufferedReader reader = new BufferedReader(new FileReader(file.getAbsolutePath()));
            String line = null;
            for (int i = 0; i < NODES_WIDTH; i++) {
                line = reader.readLine();
                for (int j = 0; j < NODES_HEIGHT; j++) {
                    //nodeList[i][j].setColor(Color.BLACK);
                    int nodeType = Character.getNumericValue(line.charAt(j));
                    switch (nodeType) {
                        case 0:
                            nodeList[i][j].setColor(Color.LIGHT_GRAY);
                            break;
                        case 1:
                            nodeList[i][j].setColor(Color.BLACK);
                            break;
                        case 2:
                            nodeList[i][j].setColor(Color.GREEN);
                            start = nodeList[i][j];
                            break;
                        case 3:
                            nodeList[i][j].setColor(Color.RED);
                            target = nodeList[i][j];
                            break; }
                    }
                }reader.close();
            }
    }// Clears the search results by resetting the color of nodes that were marked as searched.
    public void clearSearchResults() {
        for (int i = 0; i < nodeList.length; i++) {
            for (int j = 0; j < nodeList[i].length; j++) {
                if (nodeList[i][j].isSearched()) {
                    nodeList[i][j].clearNode(); }
            }
        }
        target.setColor(Color.RED);
        start.setColor(Color.GREEN);
    }//Renders graphics onto the canvas.
    public void render(Graphics2D g) {
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, WIDTH, HEIGHT);
        for (int i = 0; i < nodeList.length; i++) {
            for (int j = 0; j < nodeList[i].length; j++) {
                nodeList[i][j].render(g); }
        }
```

O(m * n)

O(m * n)

```
    }//Starts the program by creating and starting a new thread.
    public void start() {
        new Thread(this).start(); }
    public void mousePressed(MouseEvent e) {
        Node clickedNode = getNodeAt(e.getX(), e.getY());
        if (clickedNode == null) {
            return; }

        if (clickedNode.isWall()) {
            clickedNode.clearNode();
            return; }
        clickedNode.Clicked(e.getButton());
        if (clickedNode.isEnd()) {
            if (target != null) {
                target.clearNode(); }
            target = clickedNode;
        } else if (clickedNode.isStart()) {
            if (start != null) {
                start.clearNode(); }
            start = clickedNode; }
    }//Returns the start node from the grid of nodes.
    private Node getStart() {
        for (int i = 0; i < nodeList.length; i++) {
            for (int j = 0; j < nodeList[i].length; j++) {
                if (nodeList[i][j].isStart()) {
                    return nodeList[i][j];}
            }
        } return null;
    }//Returns the node at the specified coordinates on the canvas.
    public Node getNodeAt(int x, int y) {
        x -= 15;
        x /= 35;
        y -= 15;
        y /= 35;
        if (x >= 0 && y >= 0 && x < nodeList.length && y < nodeList[x].length) {
            return nodeList[x][y]; }
        return null;
    }//These methods are required by the MouseListener interface
    @Override
    public void mouseClicked(MouseEvent arg0) {}
    @Override
    public void mouseEntered(MouseEvent arg0) {}
    @Override
    public void mouseExited(MouseEvent arg0) {}
    @Override
    public void mouseReleased(MouseEvent arg0) {}
}
```

O(1)

O(1)

After doing the analysis and comparison of our algorithms:

Now let's do the complete analysis of our code:

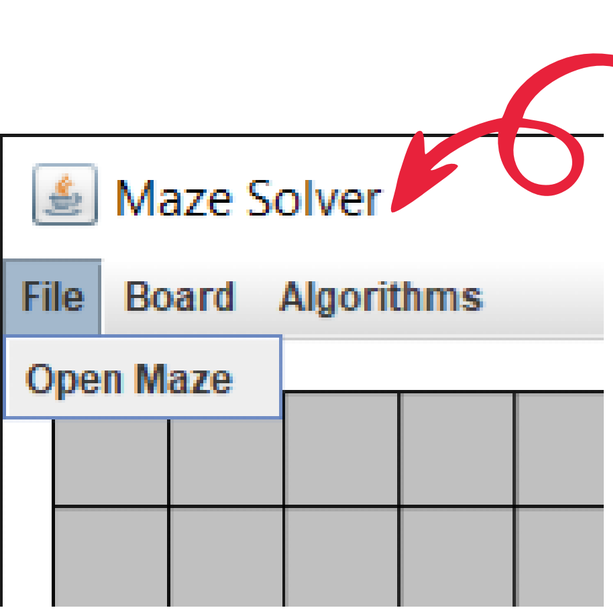
- **For Node class:**

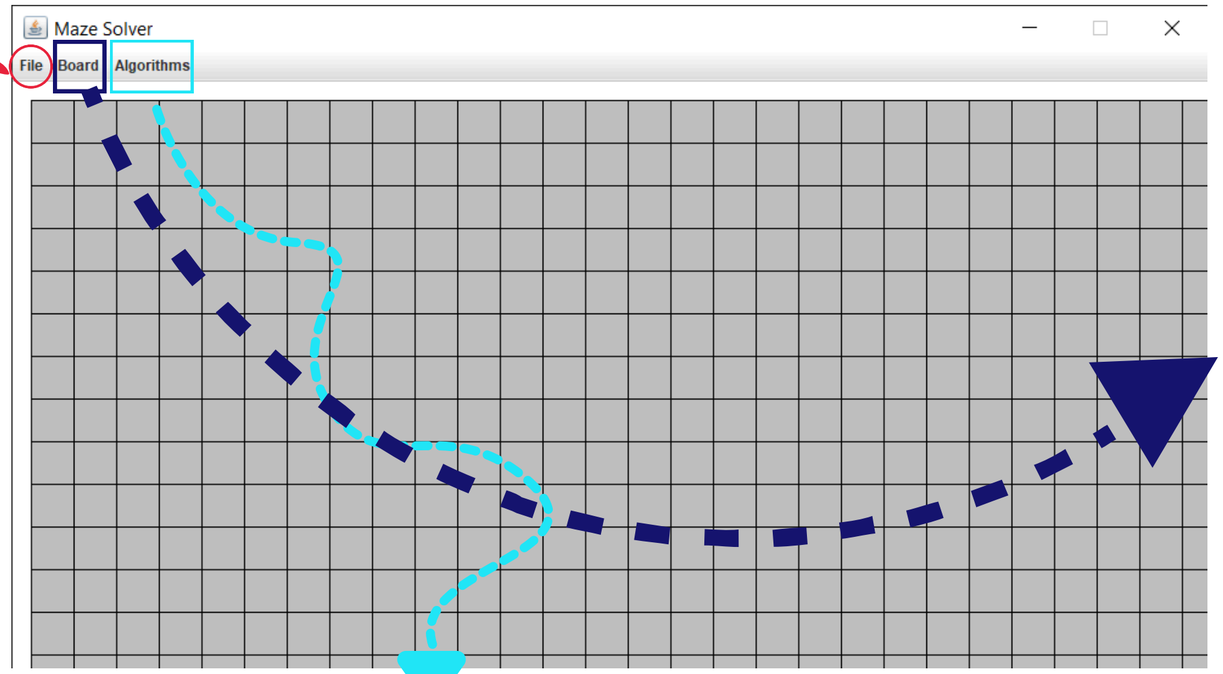
The overall time complexity of each method in the Node class is , **O(1)**, indicating constant time complexity for each operation.

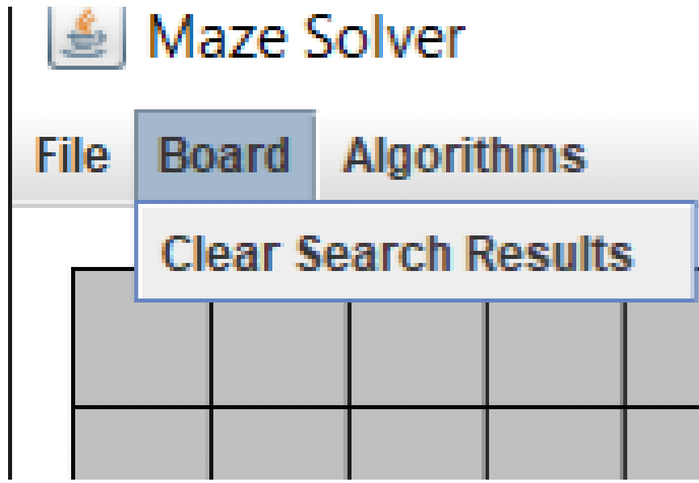
- **For Main class:**

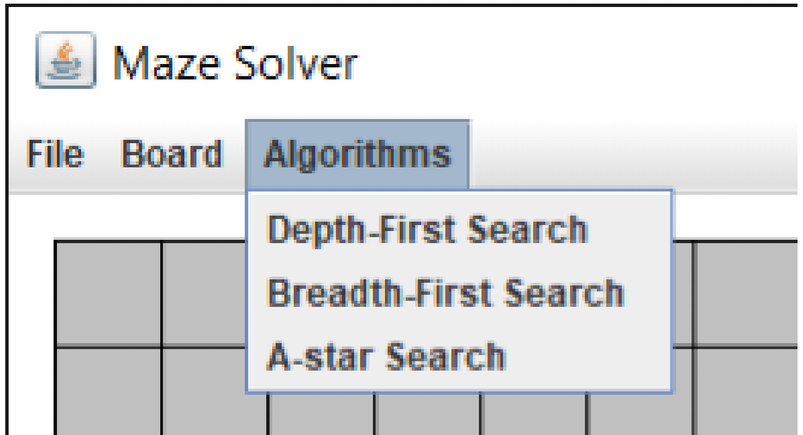
The time complexity of **O(M*N)** means that the time required for the algorithm to execute grows linearly with the size of the Maze, where M is the number of columns and N is the number of rows in the grid of nodes.

RESULT & TEST of Maze Solver System

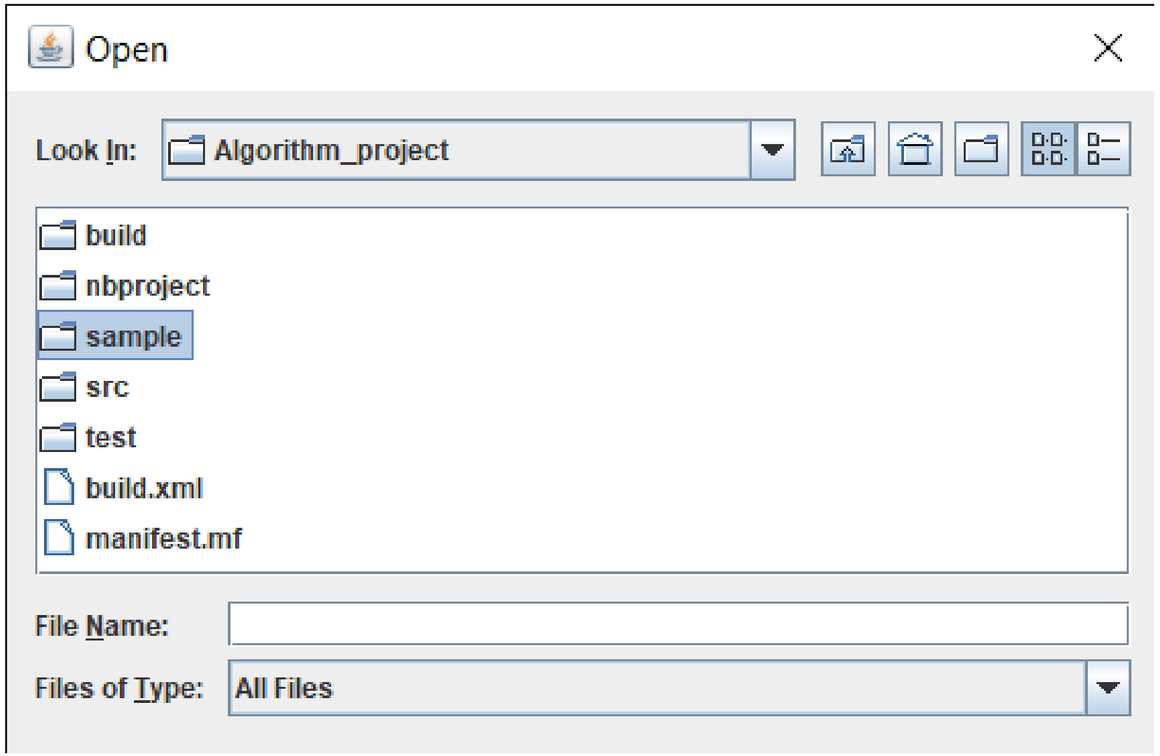


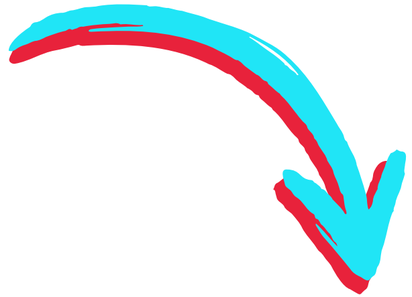


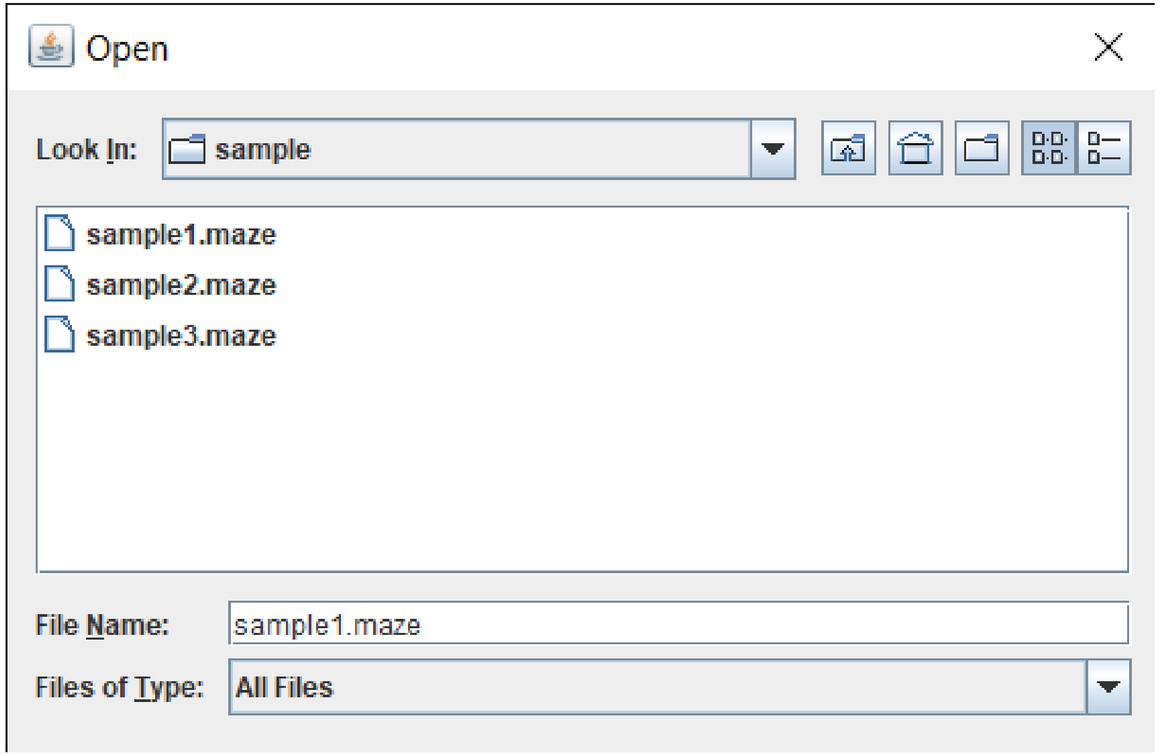




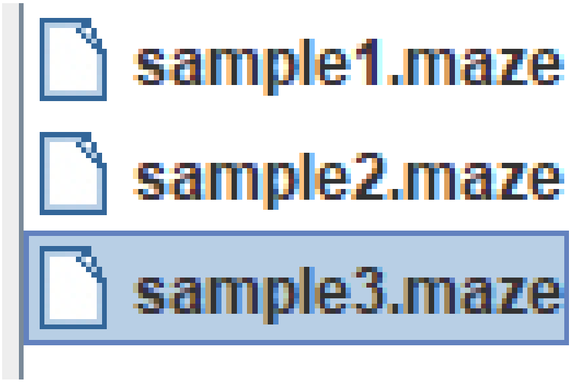
File => Open Maze

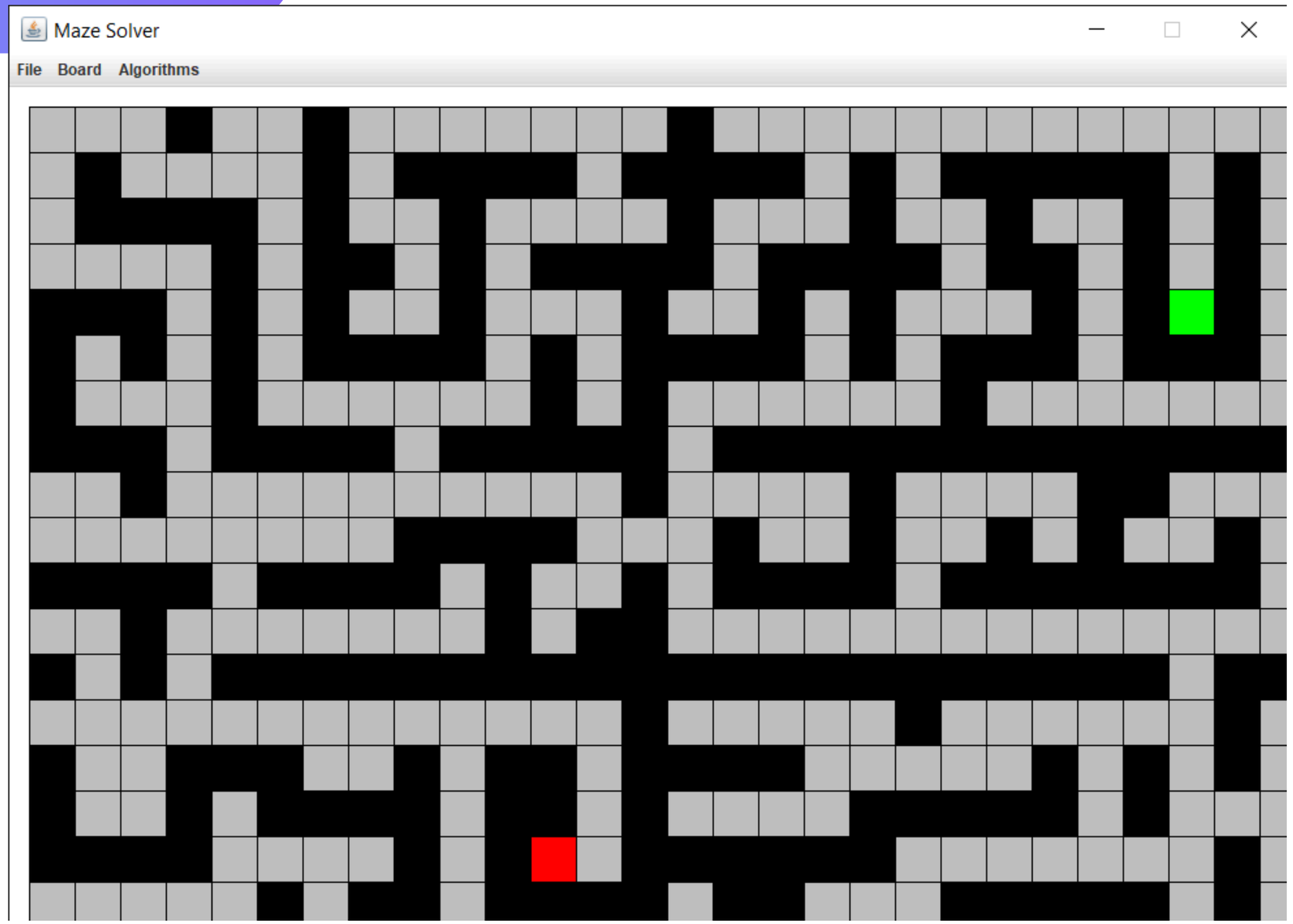




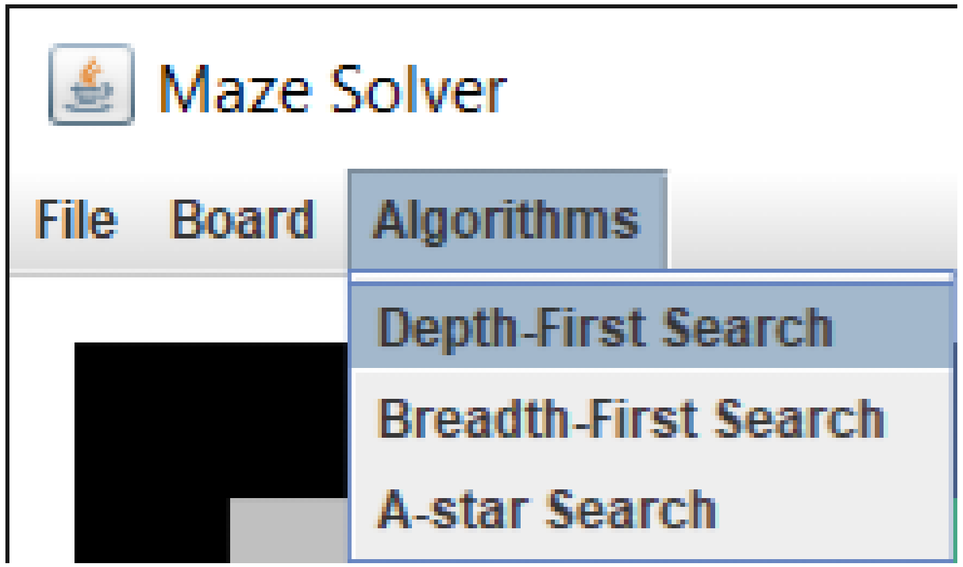


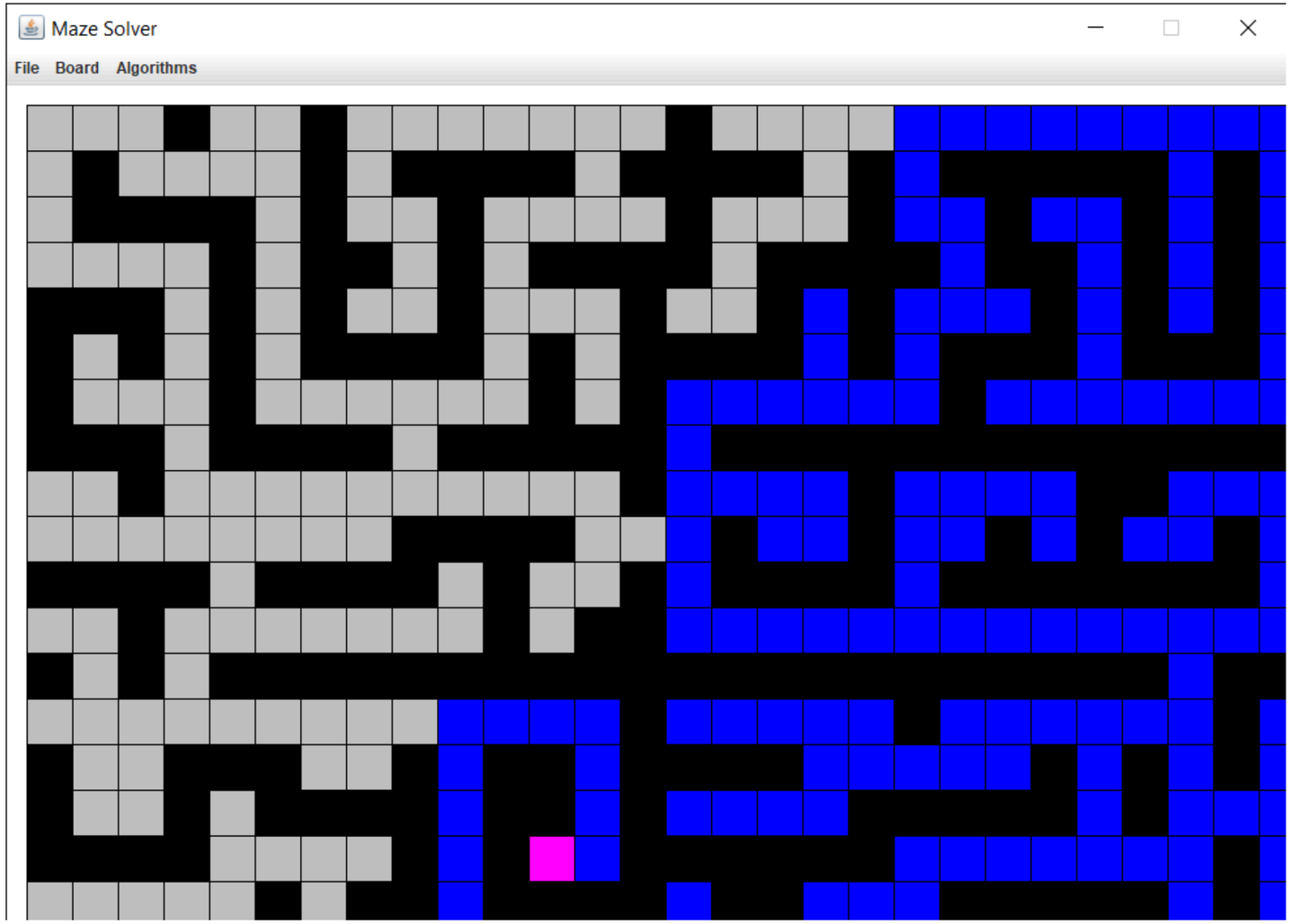
LET'S TEST EXAMPLE 3





Algorithms => Depth-First Search



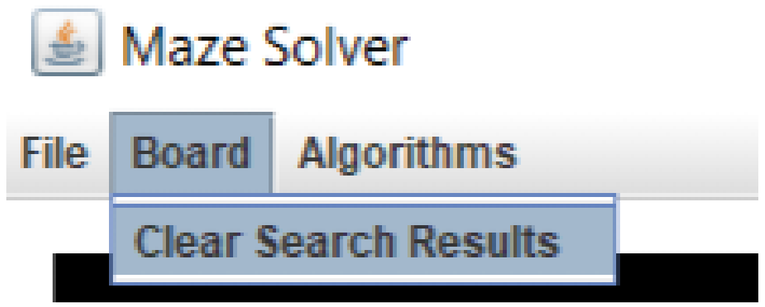


DFS Algorithm Time 0.410 ms

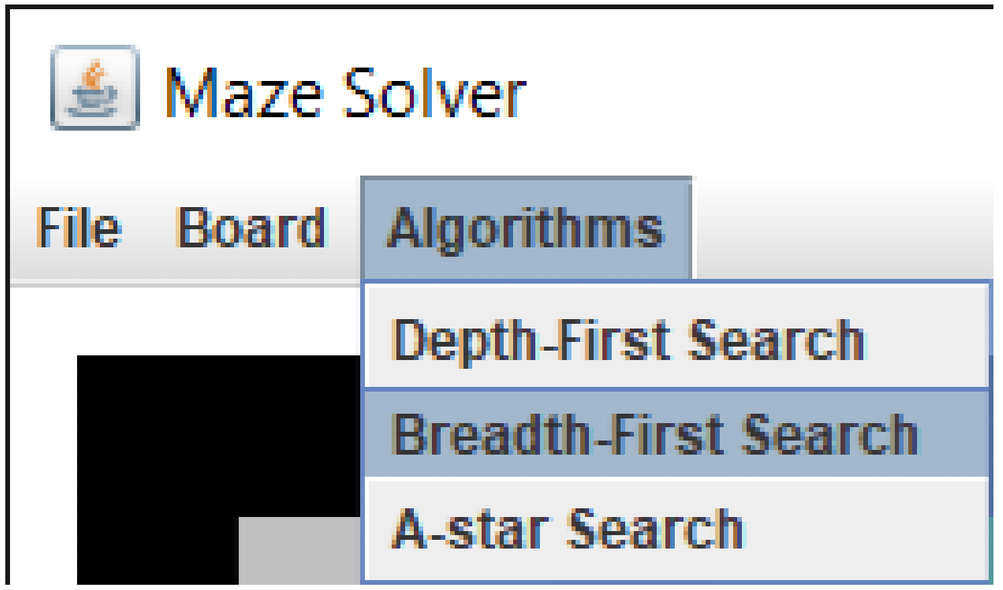
Number of nodes traversed in DFS Algorithm: 160

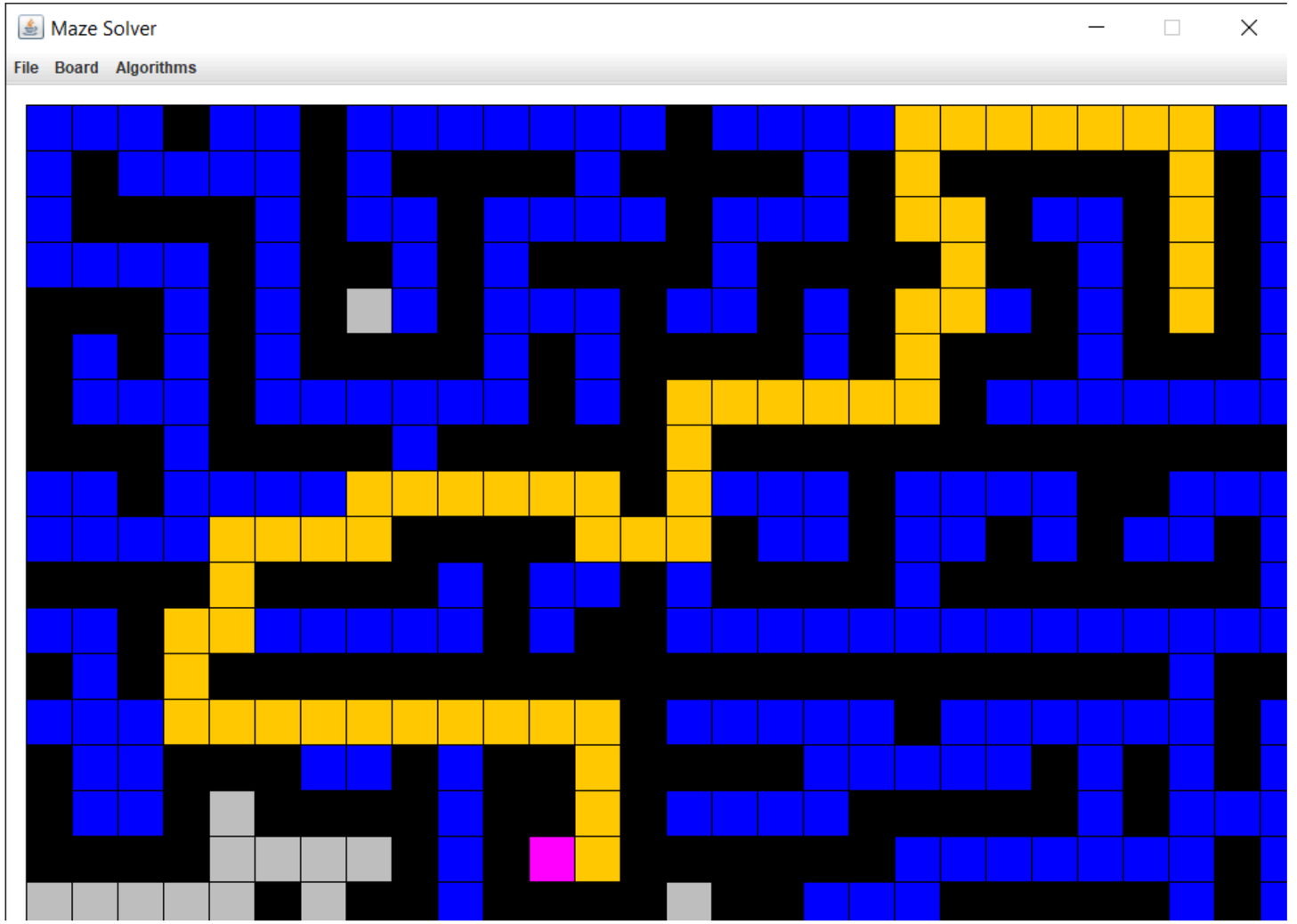
Before choose another Algorithm:

Do this step:



Then choose Algorithm



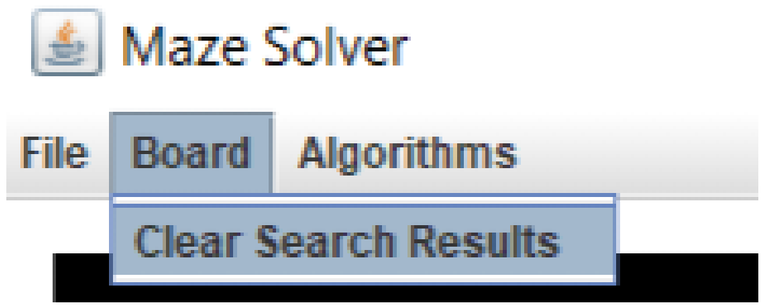


BFS Algorithm Time 0.583 ms

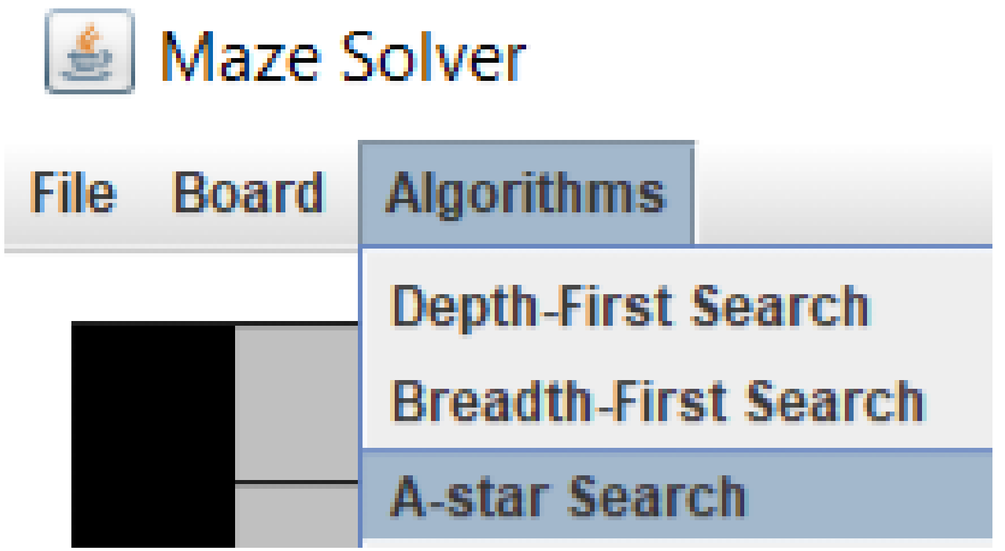
Number of nodes traversed in BFS Algorithm: 274

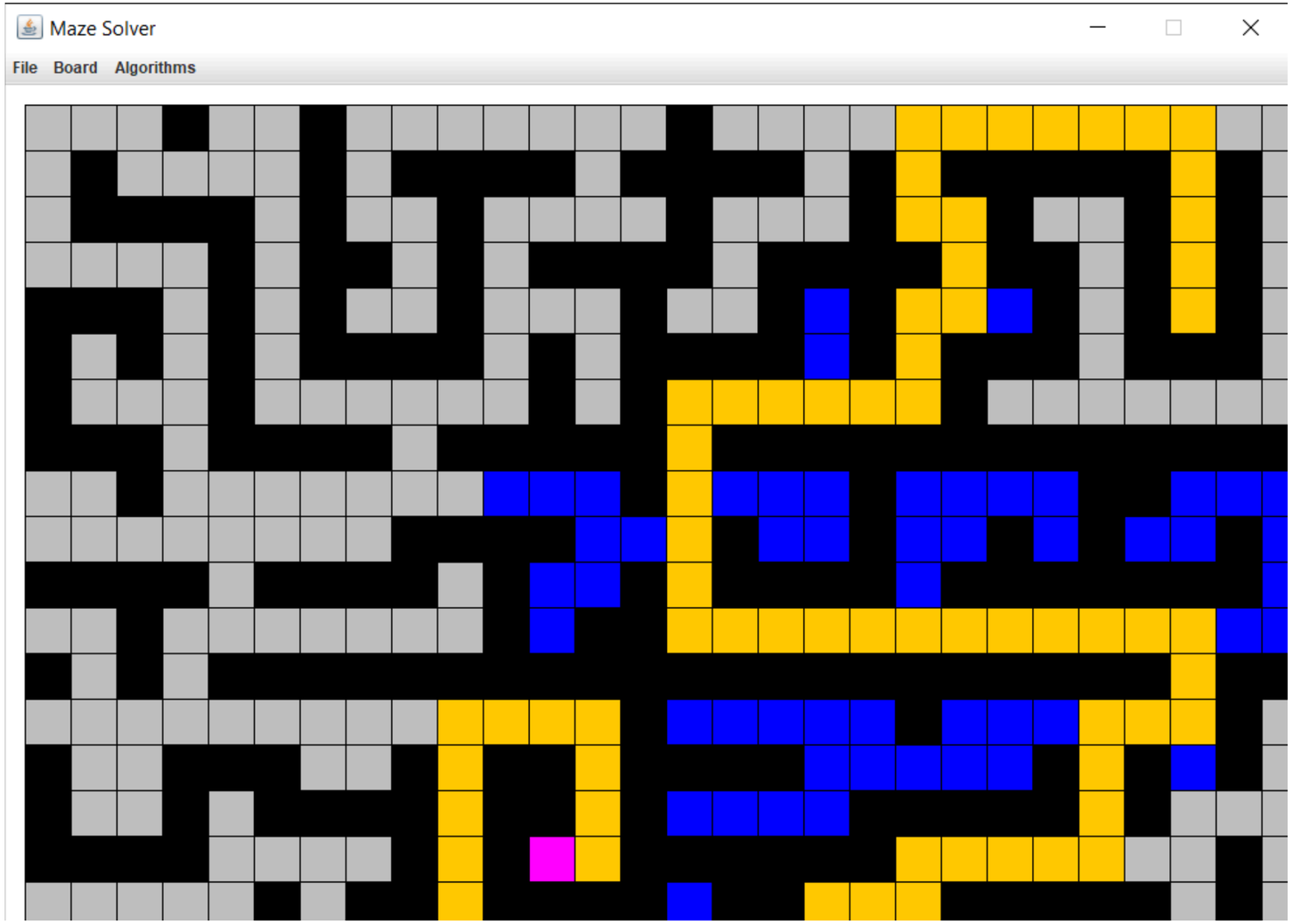
Before choose another Algorithm:

Do this step:



Then choose Algorithm

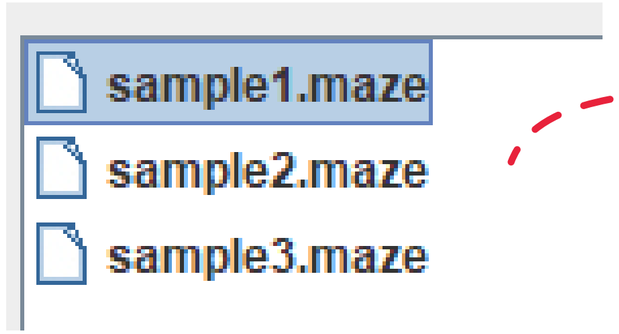


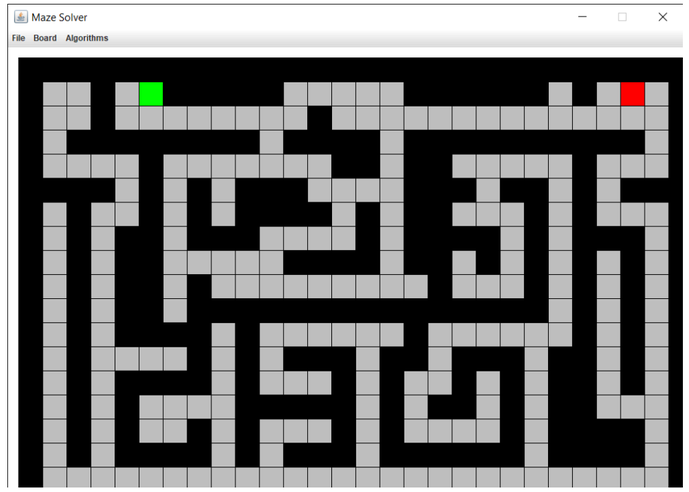


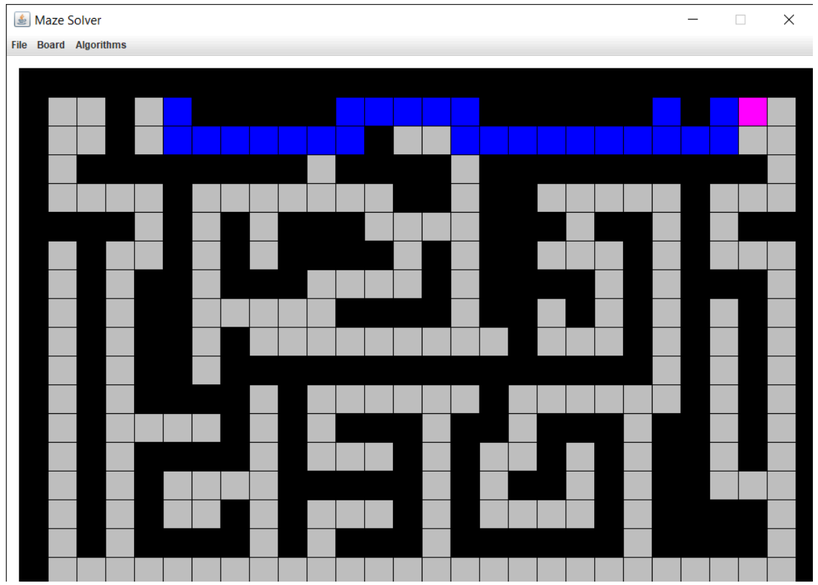
A* Algorithm Time 3.003 ms

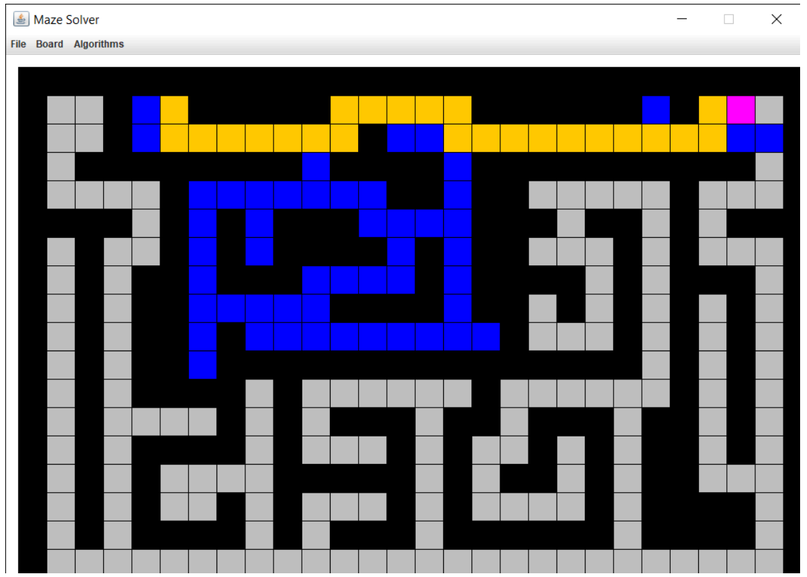
Number of nodes traversed in A* Algorithm: 128

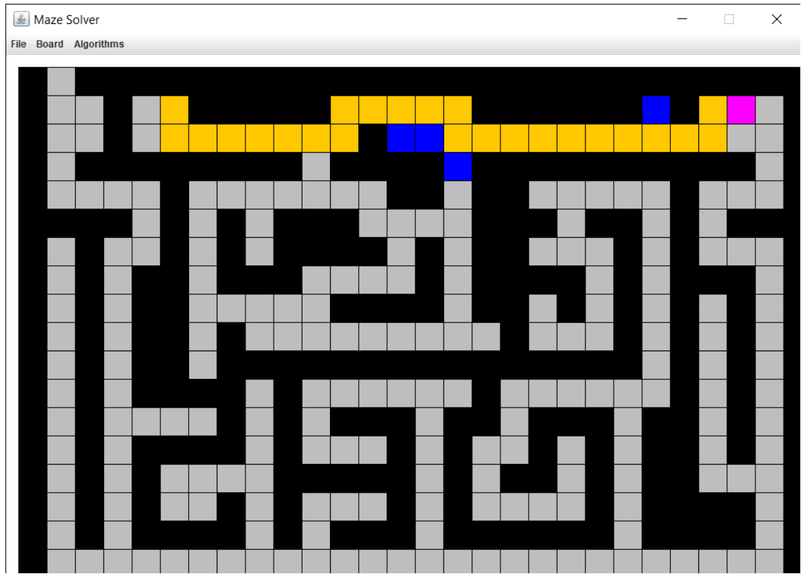
LET'S TEST EXAMPLE 1









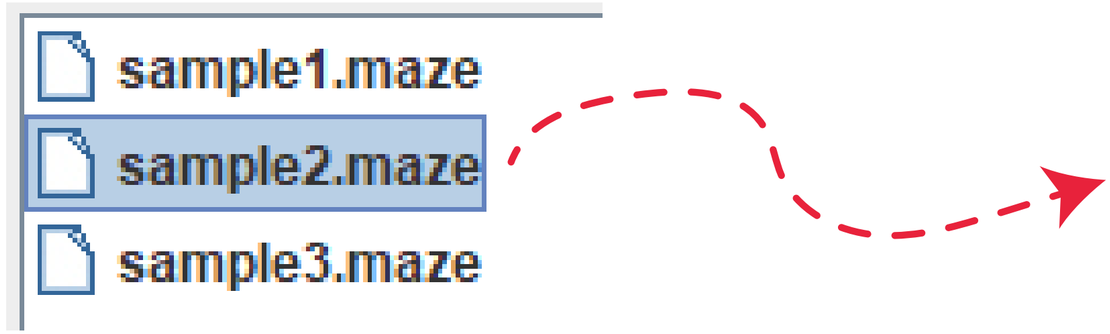


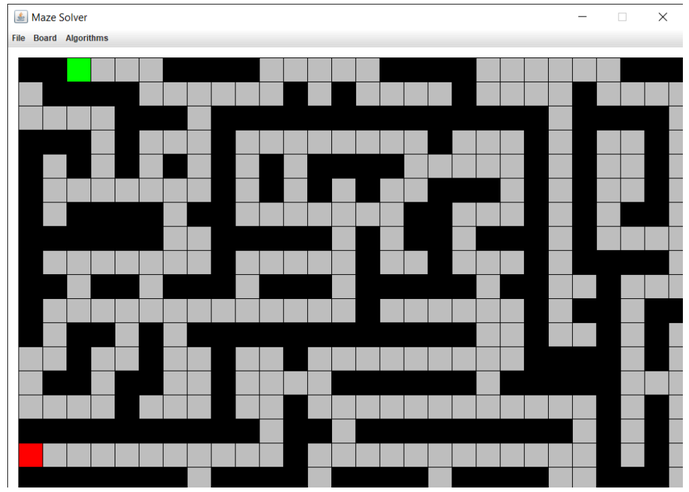
DFS Algorithm Time 0.186 ms
Number of nodes traversed in DFS Algorithm: 25

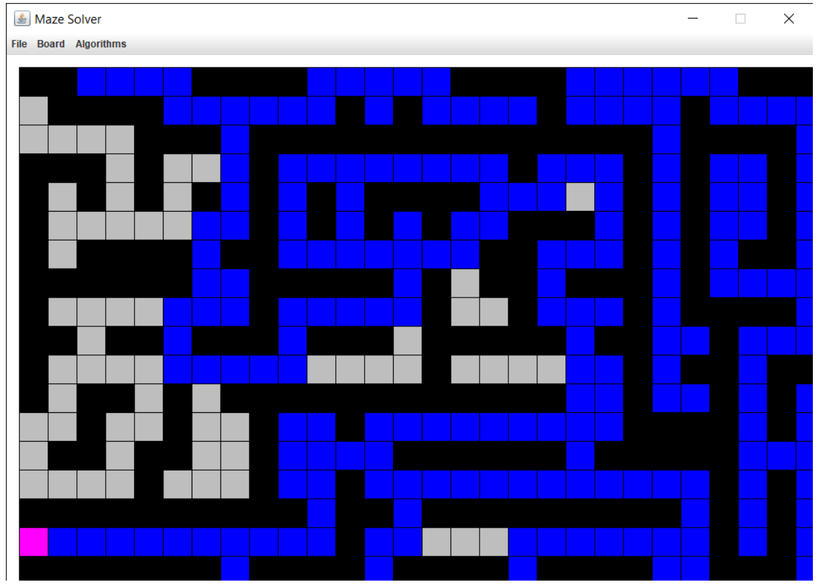
BFS Algorithm Time 0.228 ms
Number of nodes traversed in BFS Algorithm: 74

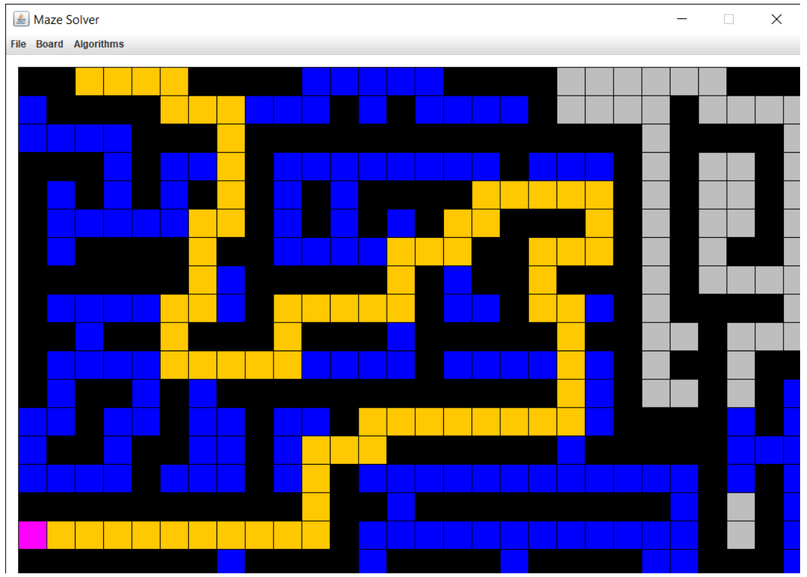
A* Algorithm Time 2.050 ms
Number of nodes traversed in A* Algorithm: 28

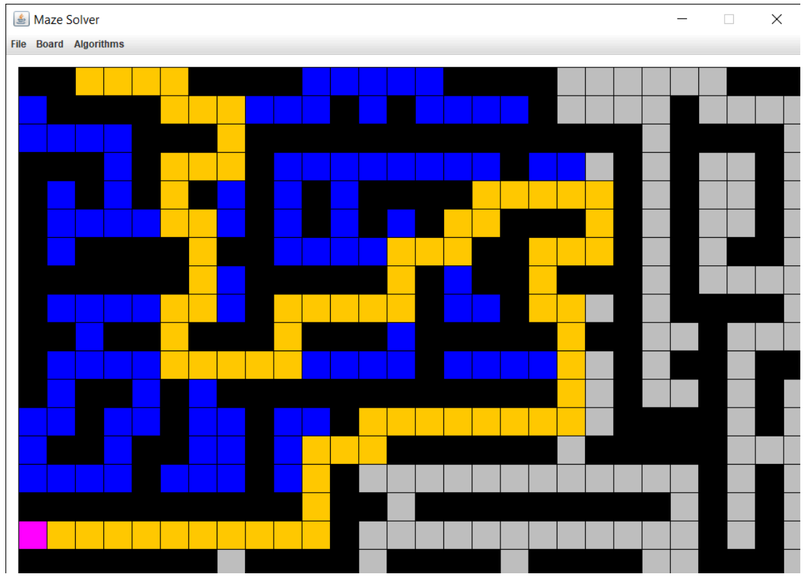
LET'S TEST EXAMPLE 2











DFS Algorithm Time 0.371 ms
Number of nodes traversed in DFS Algorithm: 236

BFS Algorithm Time 0.482 ms
Number of nodes traversed in BFS Algorithm: 239

A* Algorithm Time 3.411 ms
Number of nodes traversed in A* Algorithm: 168

Resource Page

1	https://en.m.wikipedia.org/wiki/Breadth-first_search
2	https://en.m.wikipedia.org/wiki/Depth-first_search
3	https://en.m.wikipedia.org/wiki/A*_search_algorithm
4	https://github.com/dogukanozdemir/Maze-Solver-Java


THANK
YOU