COS30019 - ASSIGNMENT 1

S1-2022

THE ROBOT NAVIGATION PROBLEM

LE BAO DUY NGUYEN

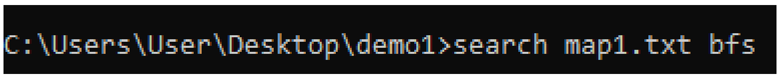
102449993

Table of Contents

1. Instructions..3
2. Introduction..4
3. Search Algorithms..4
4. Implementations..5
4.1. Un-Informed Algorithms..6
4.1.1. Depth-first Search..6
4.1.2. Breadth-first Search..7
4.1.3. Custom 1..8
4.2. Informed Algorithms..8
4.2.1. Greedy best-first Search..9
4.2.2. A Star..10
4.2.3. CUS2..11
5. Features..12
6. Research..12
7. Conclusion..13
8. Extension..13

1. Instruction

Navigate to the directory where search.exe file is placed, then use command: search.exe <map name>.txt <strategy>



```
C:\Users\User\Desktop\demo1>search map1.txt bfs
```

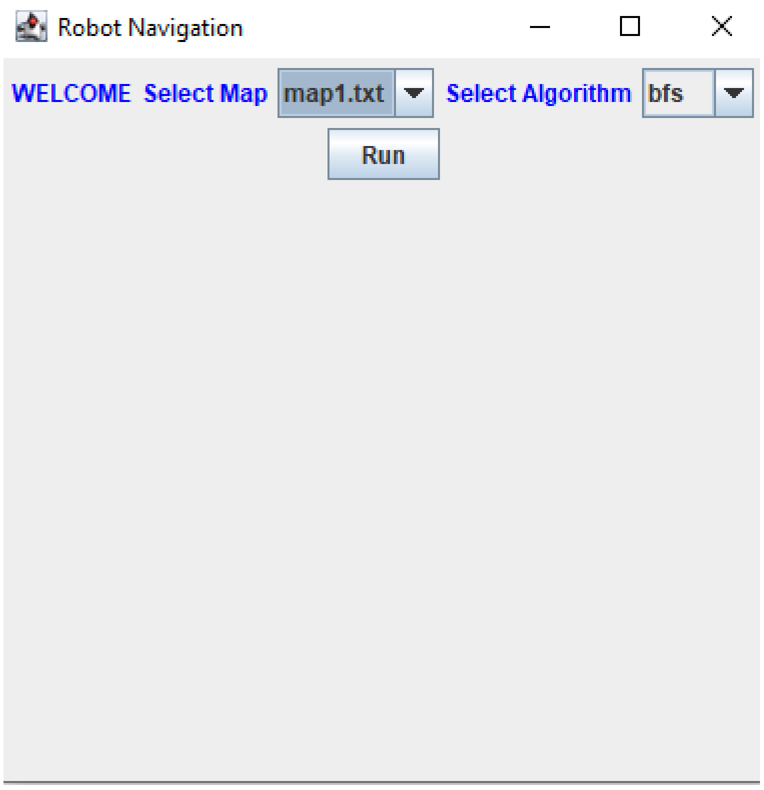
Maps:

- map1.txt
- map2.txt
- map3.txt

Search strategies:

- BFS
- DFS
- AS
- GBFS
- CUS1
- CUS2

Or run the program inside an IDE and choose the desired Map and Algorithm.



Select Map map1.txt
Select Algorithm bfs

2. Introduction

The Robot Navigation program utilises 6 different algorithms (3 informed and 3 uninformed) to find the path within some predefined environments in the form of txt files.

The environment is an NxM grid (where N > 1 and M > 1) with a number of walls occupying some cells (marked as grey cells). The robot is initially located in one of the red empty cells (Start Node) and required to find a path to visit one of the designated green cells (Destination Node) of the grid.

3. Search Algorithms

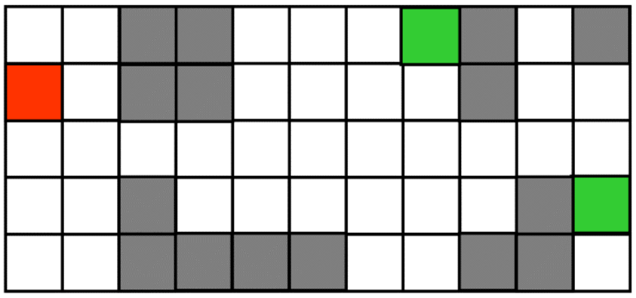
Search Type	Description	Method
Uninformed		
Depth-first Search	DFS is an algorithm for traversing or searching tree or graph data structures, that starts at the root node and explores further along each brand before backtracking. Time complexity is O(V) where V is number of nodes. In the graph, time complexity is O(V+E) where V is the number of vertexes and E is the number of edges. DFS is inefficient since it does not return the shortest path.	DFS
Breadth-first Search	BFS is an algorithm for traversing or searching tree or graph data structures, that starts at the tree root and explores all the neighbour nodes at the present depth prior to moving on to the nodes at the next depth level. Time complexity if the entire tree is traversed is O(V) where V is the number of nodes. In the graph, time complexity is O(V+E) where V is the number of vertexes and E is the number of edges. BFS guarantees the shortest path in a tree, which makes it better than DFS.	BFS
Custom 1	An uninformed method to find a path to reach the goal.	CUS1
Informed		

Greedy best-first Search	GBFS is an algorithm that always select the path that appears to be the best at execute time. It uses the heuristic cost to reach the goal from the current node to evaluate the node. GBFS expands the node which is the closest to the goal node and the closest cost is estimated by heuristic function. GBFS does not guarantee the shortest path.	GBFS
A Star (A*)	A* is an algorithm that searches for the shortest path between the initial and final state. It uses the cost to reach the goal from the current node and the cost to reach this node to evaluate the node. A* guarantees a shortest path to destination and it is efficient because it uses heuristic to find the next move.	AS
Custom 2	An informed method to find a shortest path (with least moves) to reach the goal.	CUS2

4. Implementations

[N,M] - the number of rows and the number of columns of the grid
(x1,y1) – the coordinates of the current location of the agent, the initial state
(xG1,yG1) | (xG2,yG2) | ... | (xGn,yGn), where n ≥ 1 - coordinates of the goal states
(x,y,w,h) - the leftmost top corner of the wall occupies cell (x,y) with a width of w cells and a height of h cells

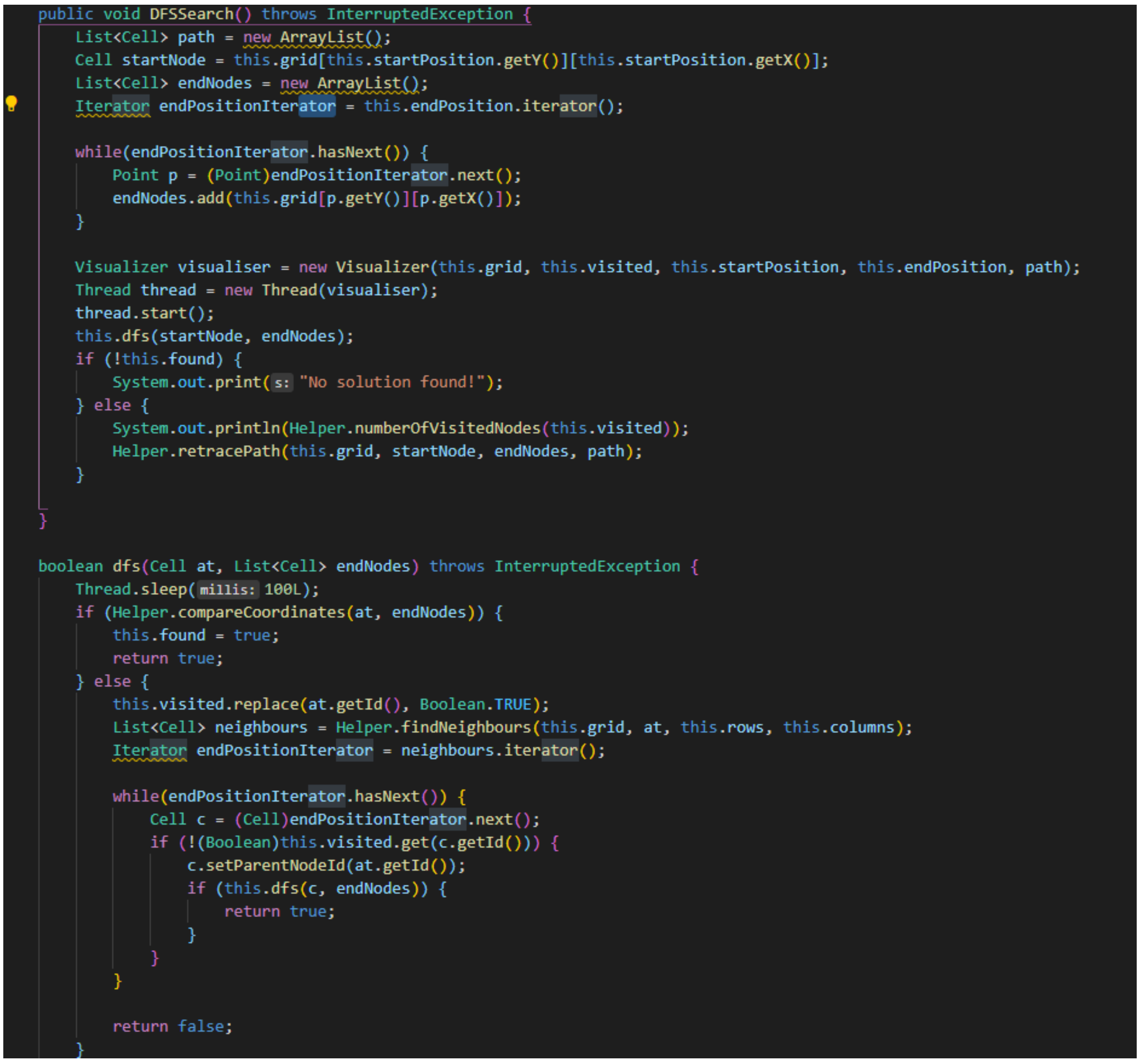
[5,11] // The grid has 5 rows and 11 columns
(0,1) // initial state of the agent – coordinates of the red cell
(7,0) | (10,3) // goal states for the agent – coordinates of the green cells
(2,0,2,2) // the square wall has the leftmost top corner occupies cell (2,0) and is 2 cells wide and 2 cell high
(8,0,1,2)
(10,0,1,1)
(2,3,1,2)
(3,4,3,1)
(9,3,1,1)
(8,4,2,1)



4.1. Un-Informed Algorithms

4.1.1. Depth-first Search

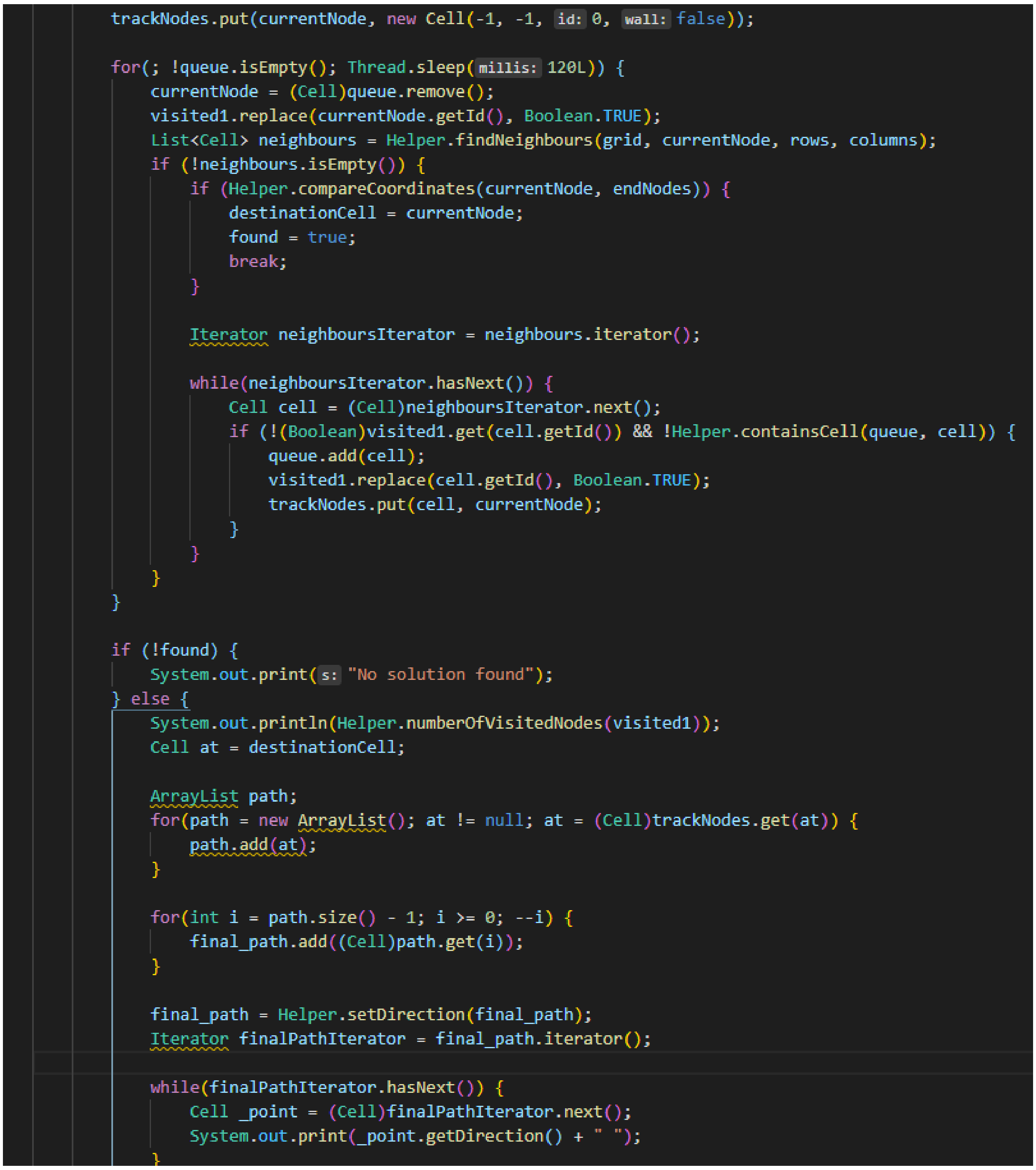
The depth-first search algorithm is built using recursion - tree builder, requiring one overload function to fit within a standard call format. initially the DFS function is called and a reference to the grid provided along with start node and destination nodes. The function generates a table to store the visited nodes and retrace back the path method to process the recessive pattern of the algorithm.



```java
public void DFSSearch() throws InterruptedException {
    List<Cell> path = new ArrayList();
    Cell startNode = this.grid[this.startPosition.getY()][this.startPosition.getX()];
    List<Cell> endNodes = new ArrayList();
    Iterator endPositionIterator = this.endPosition.iterator();

    while(endPositionIterator.hasNext()) {
        Point p = (Point)endPositionIterator.next();
        endNodes.add(this.grid[p.getY()][p.getX()]);
    }

    Visualizer visualiser = new Visualizer(this.grid, this.visited, this.startPosition, this.endPosition, path);
    Thread thread = new Thread(visualiser);
    thread.start();
    this.dfs(startNode, endNodes);
    if (!this.found) {
        System.out.print( s: "No solution found!");
    } else {
        System.out.println(Helper.numberOfVisitedNodes(this.visited));
        Helper.retracePath(this.grid, startNode, endNodes, path);
    }

}

boolean dfs(Cell at, List<Cell> endNodes) throws InterruptedException {
    Thread.sleep( millis: 100L);
    if (Helper.compareCoordinates(at, endNodes)) {
        this.found = true;
        return true;
    } else {
        this.visited.replace(at.getId(), Boolean.TRUE);
        List<Cell> neighbours = Helper.findNeighbours(this.grid, at, this.rows, this.columns);
        Iterator endPositionIterator = neighbours.iterator();

        while(endPositionIterator.hasNext()) {
            Cell c = (Cell)endPositionIterator.next();
            if (!(Boolean)this.visited.get(c.getId())) {
                c.setParentNodeId(at.getId());
                if (this.dfs(c, endNodes)) {
                    return true;
                }
            }
        }

        return false;
    }
```

4.1.2. Breadth-first Search

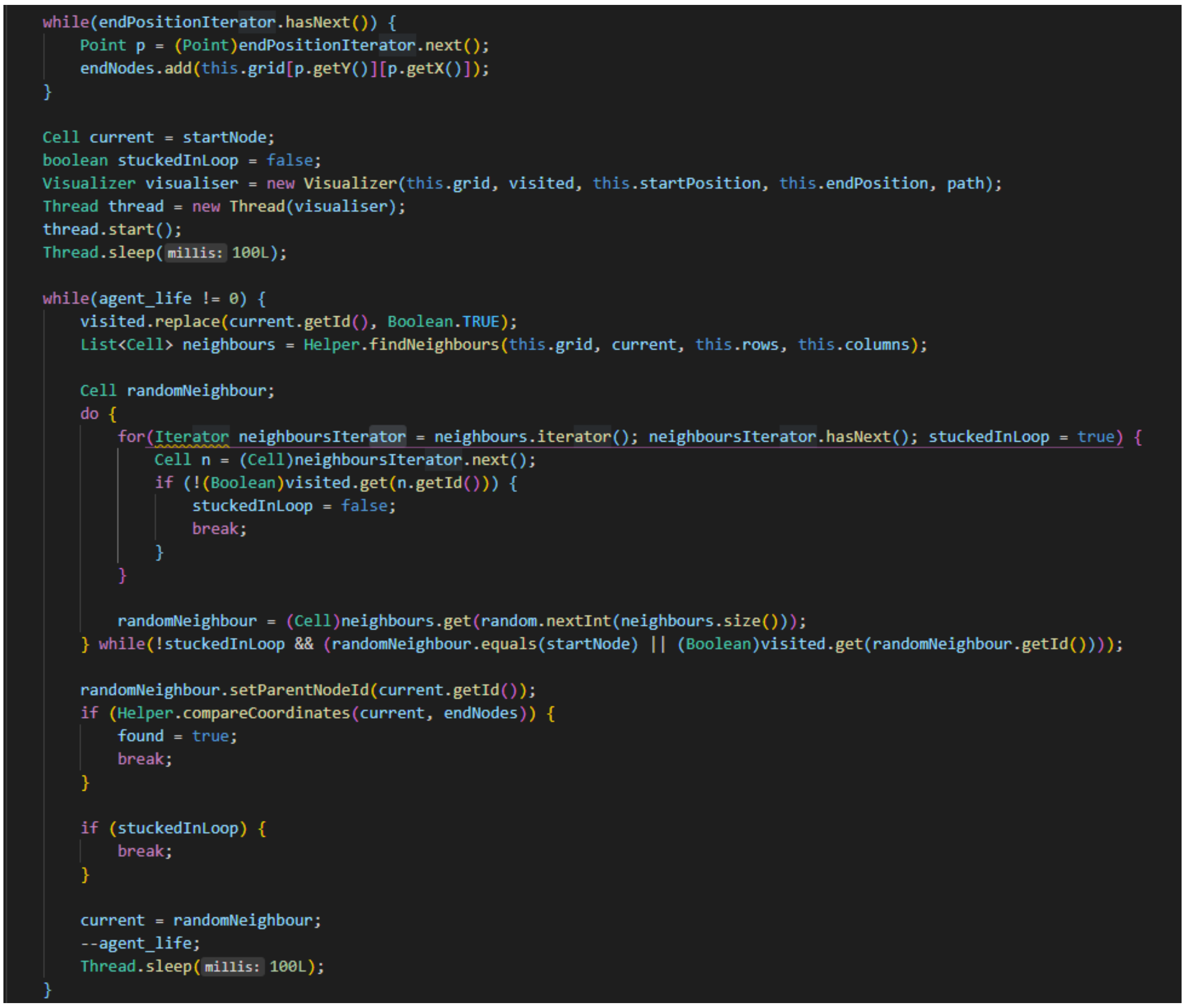
The BFS program picks a node and enqueue all its adjacent nodes into a queue. Dequeued nodes are marked as visited, whose adjacent nodes will then be enqueued. This process is repeated until the queue is empty or the goal is met. This algorithm can be stuck in an infinite loop if a node is revisited and was not marked as visited before. Hence, prevent exploring nodes that are visited by marking them as visited.



```
        trackNodes.put(currentNode, new Cell(-1, -1, id: 0, wall: false));

        for(; !queue.isEmpty(); Thread.sleep(millis: 120L)) {
            currentNode = (Cell)queue.remove();
            visited1.replace(currentNode.getId(), Boolean.TRUE);
            List<Cell> neighbours = Helper.findNeighbours(grid, currentNode, rows, columns);
            if (!neighbours.isEmpty()) {
                if (Helper.compareCoordinates(currentNode, endNodes)) {
                    destinationCell = currentNode;
                    found = true;
                    break;
                }

                Iterator neighboursIterator = neighbours.iterator();

                while(neighboursIterator.hasNext()) {
                    Cell cell = (Cell)neighboursIterator.next();
                    if (!(Boolean)visited1.get(cell.getId()) && !Helper.containsCell(queue, cell)) {
                        queue.add(cell);
                        visited1.replace(cell.getId(), Boolean.TRUE);
                        trackNodes.put(cell, currentNode);
                    }
                }
            }
        }

        if (!found) {
            System.out.print(s: "No solution found");
        } else {
            System.out.println(Helper.numberOfVisitedNodes(visited1));
            Cell at = destinationCell;

            ArrayList path;
            for(path = new ArrayList(); at != null; at = (Cell)trackNodes.get(at)) {
                path.add(at);
            }

            for(int i = path.size() - 1; i >= 0; --i) {
                final_path.add((Cell)path.get(i));
            }

            final_path = Helper.setDirection(final_path);
            Iterator finalPathIterator = final_path.iterator();

            while(finalPathIterator.hasNext()) {
                Cell _point = (Cell)finalPathIterator.next();
                System.out.print(_point.getDirection() + " ");
```

4.1.3. Custom 1

The custom un-informed search algorithm is based on the array sorting method known as bogo-sort, which is an ineffective search algorithm. CUS1 randomly generating permutations of the array and checking whether it is sorted or not. This is rather very luck-based, which most of the time can cause loops and normally just a demonstration on how ineffective algorithm behaves different from others.



```
while(endPositionIterator.hasNext()) {
    Point p = (Point)endPositionIterator.next();
    endNodes.add(this.grid[p.getY()][p.getX()]);
}

Cell current = startNode;
boolean stuckedInLoop = false;
Visualizer visualiser = new Visualizer(this.grid, visited, this.startPosition, this.endPosition, path);
Thread thread = new Thread(visualiser);
thread.start();
Thread.sleep( millis: 100L);

while(agent_life != 0) {
    visited.replace(current.getId(), Boolean.TRUE);
    List<Cell> neighbours = Helper.findNeighbours(this.grid, current, this.rows, this.columns);

    Cell randomNeighbour;
    do {
        for(Iterator neighboursIterator = neighbours.iterator(); neighboursIterator.hasNext(); stuckedInLoop = true) {
            Cell n = (Cell)neighboursIterator.next();
            if (!(Boolean)visited.get(n.getId())) {
                stuckedInLoop = false;
                break;
            }
        }

        randomNeighbour = (Cell)neighbours.get(random.nextInt(neighbours.size()));
    } while(!stuckedInLoop && (randomNeighbour.equals(startNode) || (Boolean)visited.get(randomNeighbour.getId())));

    randomNeighbour.setParentNodeId(current.getId());
    if (Helper.compareCoordinates(current, endNodes)) {
        found = true;
        break;
    }

    if (stuckedInLoop) {
        break;
    }

    current = randomNeighbour;
    --agent_life;
    Thread.sleep( millis: 100L);
}
```

4.2. Informed Algorithms

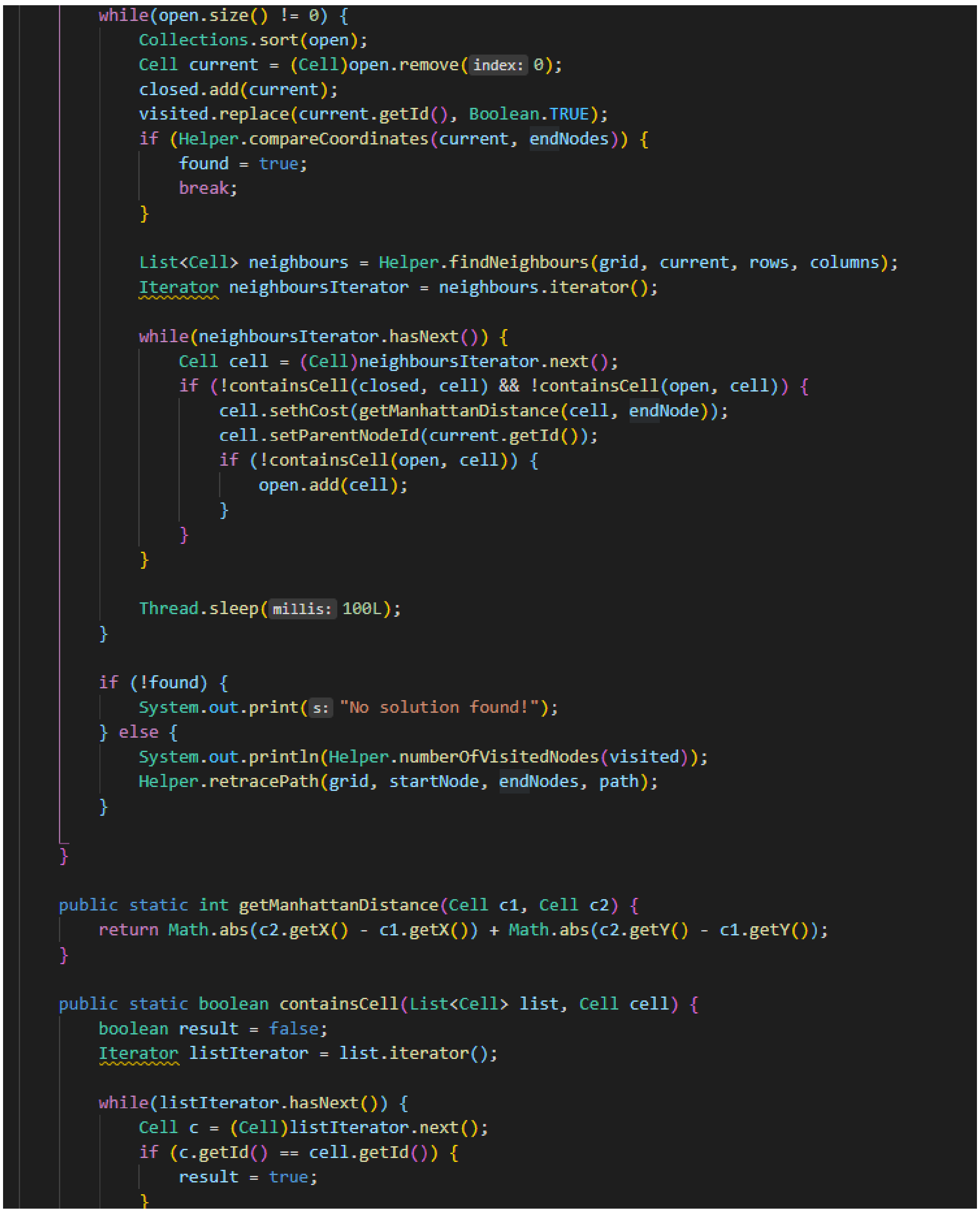
- Depth-first Search
- Breadth-first Search
- Custom 2

The greedy best-first search and A* search algorithms require a extra step for cost calculation method to work correctly. Calculating the cost based on the positioning of a to and from node on the grid.

Both functions (A* and GBFS) use a well-known distance method known as Manhattan Distance, the distance required to travel along the x and y paths.

4.2.1. Greedy best-first Search

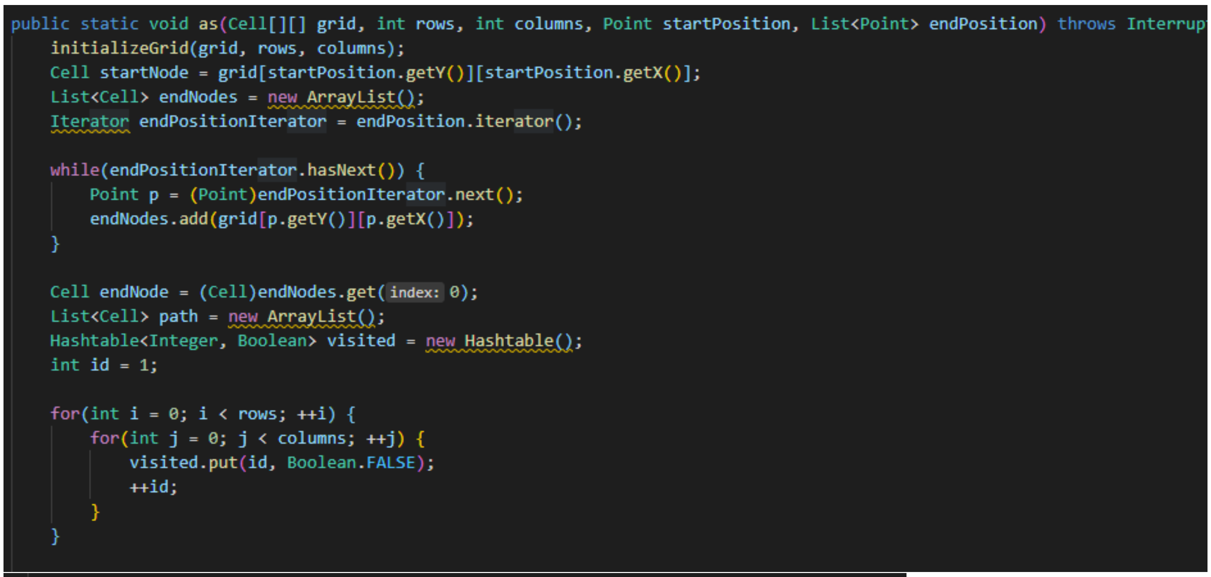
The Greedy Best First Search uses Manhattan distance to calculate heuristic. Greedy best-first selects the path that appears to be the best at run-time.



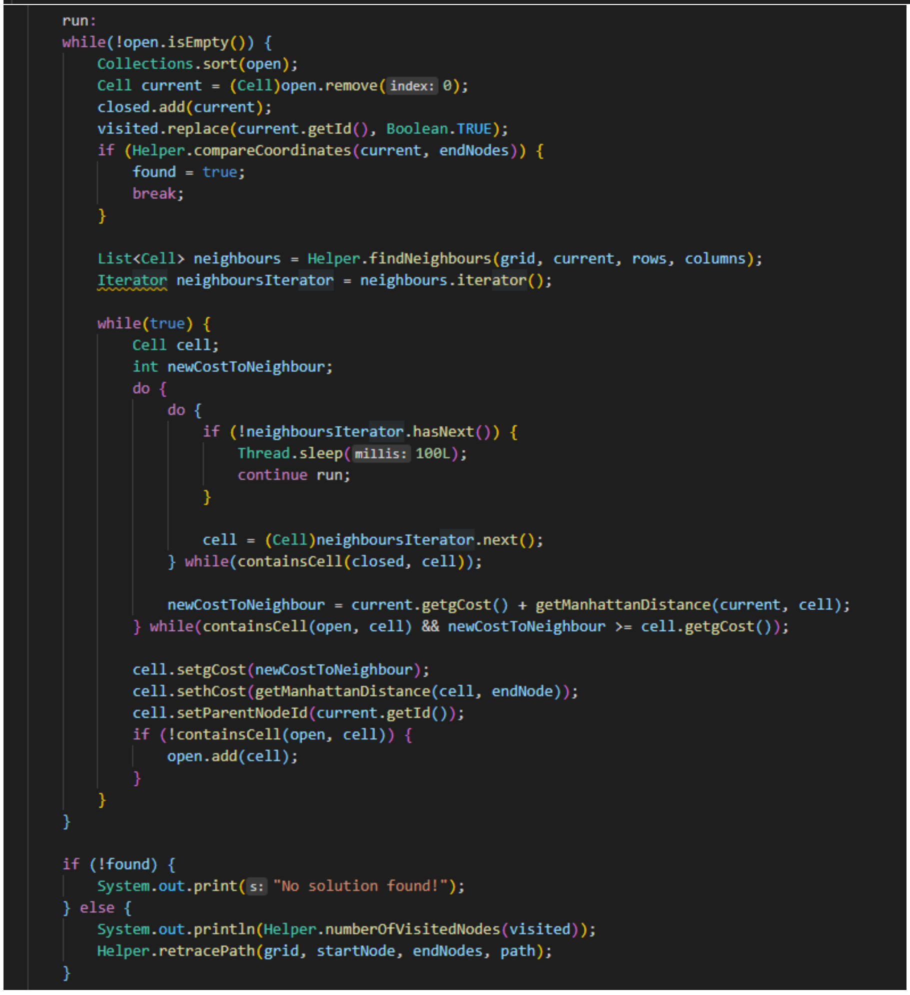
```java
        while(open.size() != 0) {
            Collections.sort(open);
            Cell current = (Cell)open.remove( index: 0);
            closed.add(current);
            visited.replace(current.getId(), Boolean.TRUE);
            if (Helper.compareCoordinates(current, endNodes)) {
                found = true;
                break;
            }

            List<Cell> neighbours = Helper.findNeighbours(grid, current, rows, columns);
            Iterator neighboursIterator = neighbours.iterator();

            while(neighboursIterator.hasNext()) {
                Cell cell = (Cell)neighboursIterator.next();
                if (!containsCell(closed, cell) && !containsCell(open, cell)) {
                    cell.sethCost(getManhattanDistance(cell, endNode));
                    cell.setParentNodeId(current.getId());
                    if (!containsCell(open, cell)) {
                        open.add(cell);
                    }
                }
            }

            Thread.sleep( millis: 100L);
        }

        if (!found) {
            System.out.print( s: "No solution found!");
        } else {
            System.out.println(Helper.numberOfVisitedNodes(visited));
            Helper.retracePath(grid, startNode, endNodes, path);
        }

    }

    public static int getManhattanDistance(Cell c1, Cell c2) {
        return Math.abs(c2.getX() - c1.getX()) + Math.abs(c2.getY() - c1.getY());
    }

    public static boolean containsCell(List<Cell> list, Cell cell) {
        boolean result = false;
        Iterator listIterator = list.iterator();

        while(listIterator.hasNext()) {
            Cell c = (Cell)listIterator.next();
            if (c.getId() == cell.getId()) {
                result = true;
            }
```

4.2.2. A Star

The A* search algorithm uses both the cost to travel from the starting point to the node, as well as the cost to travel from the node to the goal to assess the node to determine the best path.



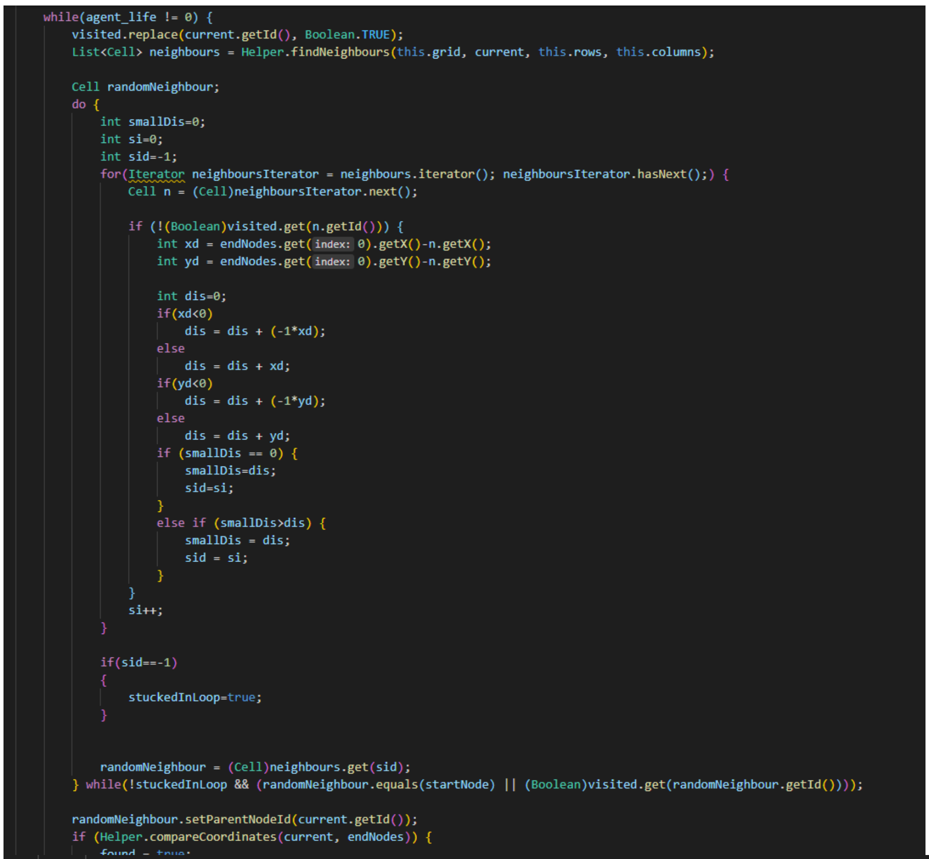
```java
public static void as(Cell[][] grid, int rows, int columns, Point startPosition, List<Point> endPosition) throws Interrup
    initializeGrid(grid, rows, columns);
    Cell startNode = grid[startPosition.getY()][startPosition.getX()];
    List<Cell> endNodes = new ArrayList();
    Iterator endPositionIterator = endPosition.iterator();

    while(endPositionIterator.hasNext()) {
        Point p = (Point)endPositionIterator.next();
        endNodes.add(grid[p.getY()][p.getX()]);
    }

    Cell endNode = (Cell)endNodes.get( index: 0);
    List<Cell> path = new ArrayList();
    Hashtable<Integer, Boolean> visited = new Hashtable();
    int id = 1;

    for(int i = 0; i < rows; ++i) {
        for(int j = 0; j < columns; ++j) {
            visited.put(id, Boolean.FALSE);
            ++id;
        }
    }
```



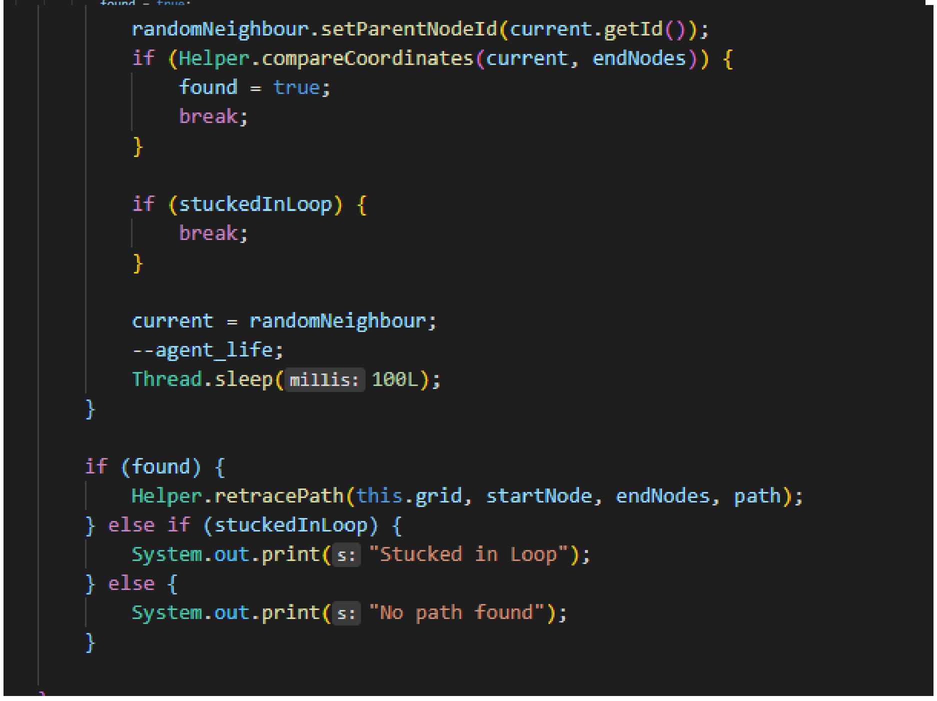
```java
    run:
    while(!open.isEmpty()) {
        Collections.sort(open);
        Cell current = (Cell)open.remove( index: 0);
        closed.add(current);
        visited.replace(current.getId(), Boolean.TRUE);
        if (Helper.compareCoordinates(current, endNodes)) {
            found = true;
            break;
        }

        List<Cell> neighbours = Helper.findNeighbours(grid, current, rows, columns);
        Iterator neighboursIterator = neighbours.iterator();

        while(true) {
            Cell cell;
            int newCostToNeighbour;
            do {
                do {
                    if (!neighboursIterator.hasNext()) {
                        Thread.sleep( millis: 100L);
                        continue run;
                    }

                    cell = (Cell)neighboursIterator.next();
                } while(containsCell(closed, cell));

                newCostToNeighbour = current.getgCost() + getManhattanDistance(current, cell);
            } while(containsCell(open, cell) && newCostToNeighbour >= cell.getgCost());

            cell.setgCost(newCostToNeighbour);
            cell.sethCost(getManhattanDistance(cell, endNode));
            cell.setParentNodeId(current.getId());
            if (!containsCell(open, cell)) {
                open.add(cell);
            }
        }
    }

    if (!found) {
        System.out.print( s: "No solution found!");
    } else {
        System.out.println(Helper.numberOfVisitedNodes(visited));
        Helper.retracePath(grid, startNode, endNodes, path);
    }
```

4.2.3. CUS2

CUS2 does iteration all the current nodes, it would check all neighbours in order and calculate the distance from the starting node to each of them, then select the shortest one.



```java
while(agent_life != 0) {
    visited.replace(current.getId(), Boolean.TRUE);
    List<Cell> neighbours = Helper.findNeighbours(this.grid, current, this.rows, this.columns);

    Cell randomNeighbour;
    do {
        int smallDis=0;
        int si=0;
        int sid=-1;
        for(Iterator neighboursIterator = neighbours.iterator(); neighboursIterator.hasNext();) {
            Cell n = (Cell)neighboursIterator.next();

            if (!(Boolean)visited.get(n.getId())) {
                int xd = endNodes.get(index: 0).getX()-n.getX();
                int yd = endNodes.get(index: 0).getY()-n.getY();

                int dis=0;
                if(xd<0)
                    dis = dis + (-1*xd);
                else
                    dis = dis + xd;
                if(yd<0)
                    dis = dis + (-1*yd);
                else
                    dis = dis + yd;
                if (smallDis == 0) {
                    smallDis=dis;
                    sid=si;
                }
                else if (smallDis>dis) {
                    smallDis = dis;
                    sid = si;
                }
            }
            si++;
        }

        if(sid==-1)
        {
            stuckedInLoop=true;
        }


        randomNeighbour = (Cell)neighbours.get(sid);
    } while(!stuckedInLoop && (randomNeighbour.equals(startNode) || (Boolean)visited.get(randomNeighbour.getId())));

    randomNeighbour.setParentNodeId(current.getId());
    if (Helper.compareCoordinates(current, endNodes)) {
```



```java
        randomNeighbour.setParentNodeId(current.getId());
        if (Helper.compareCoordinates(current, endNodes)) {
            found = true;
            break;
        }

        if (stuckedInLoop) {
            break;
        }

        current = randomNeighbour;
        --agent_life;
        Thread.sleep(millis: 100L);
    }

    if (found) {
        Helper.retracePath(this.grid, startNode, endNodes, path);
    } else if (stuckedInLoop) {
        System.out.print(s: "Stucked in Loop");
    } else {
        System.out.print(s: "No path found");
    }
```

5. Features

Visualiser: Visualiser using awt and swing to visualise the current state of the search as well as animate the searching nodes and path.

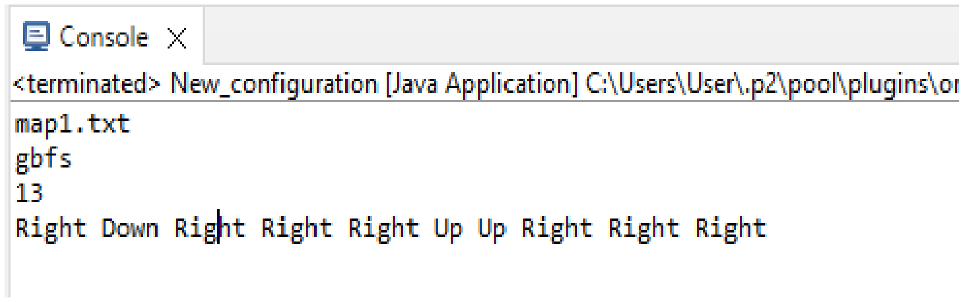
Output is in the required format.

filename

method

number_of_nodes

path



```
<terminated> New_configuration [Java Application] C:\Users\User\.p2\pool\plugins\or
map1.txt
gbfs
13
Right Down Right Right Right Up Up Right Right Right
```

6. Research

Bin, M., 2021. *Graph Traversal in Python: BFS,DFS,Dijkstra,A-star parallel comparision*. [online] Medium. Available at: <https://medium.com/nerd-for-tech/graph-traversal-in-python-bfs-dfs-dijkstra-a-star-parallel-comparision-dd4132ec323a> [Accessed 14 April 2022].

www.javatpoint.com. n.d. *Informed Search Algorithms in AI - Javatpoint*. [online] Available at: <https://www.javatpoint.com/ai-informed-search-algorithms> [Accessed 14 April 2022].

Stack Overflow. 2010. *Are there any worse sorting algorithms than Bogosort (a.k.a Monkey Sort)?*. [online] Available at: <https://stackoverflow.com/questions/2609857/are-there-any-worse-sorting-algorithms-than-bogosort-a-k-a-monkey-sort> [Accessed 18 April 2022].

www.javatpoint.com. n.d. *AWT and Swing in Java - Javatpoint*. [online] Available at: <https://www.javatpoint.com/awt-and-swing-in-java> [Accessed 18 April 2022].

Oracle.com. n.d. *Painting in AWT and Swing*. [online] Available at: <https://www.oracle.com/java/technologies/painting.html> [Accessed 18 April 2022].

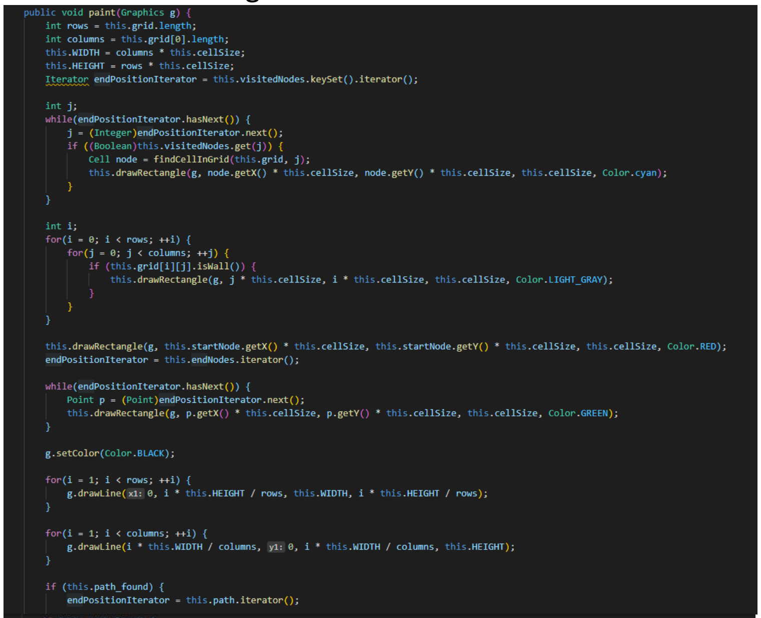
7. Conclusion

For the standard algorithms of similar test cases, A* would be recommended as it finds the shortest and quickest path to the goal. DFS would be the least recommended as it not only the longest path, but the calculation of nodes is time and resource consuming that it is not desirable.

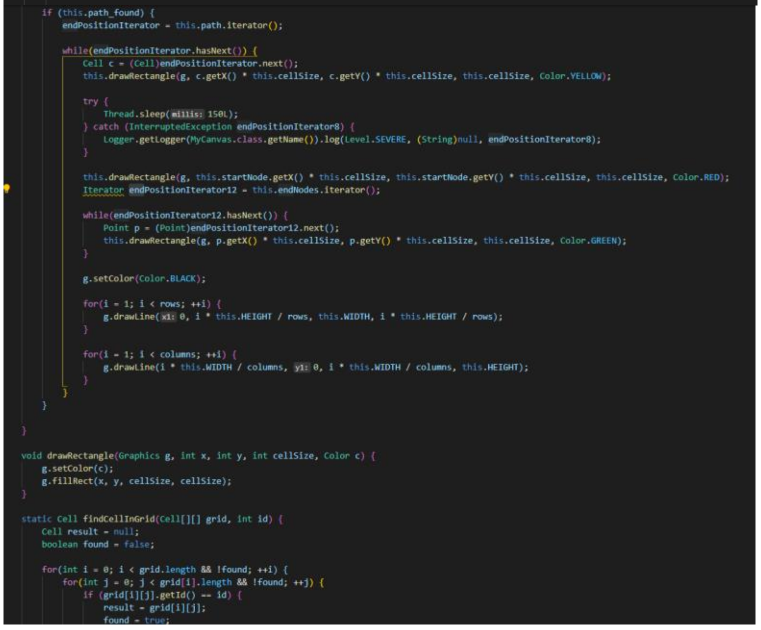
CUS1 is an example of how algorithm can be badly designed and holds little value. CUS2 performs an informed method that acts as brute-force and can provide the path to goal quickest as possible.

8. Extension

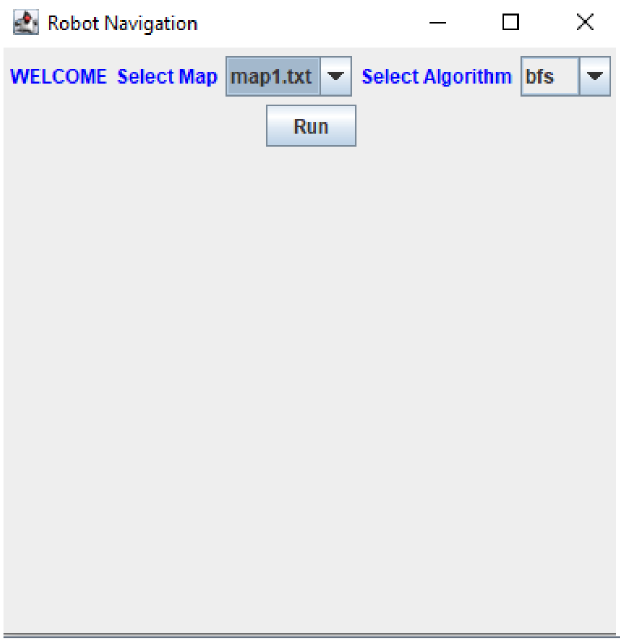
GUI Visualiser using JAVA.AWT and SWING to visualise the algorithms.



```java
public void paint(Graphics g) {
    int rows = this.grid.length;
    int columns = this.grid[0].length;
    this.WIDTH = columns * this.cellSize;
    this.HEIGHT = rows * this.cellSize;
    Iterator endPositionIterator = this.visitedNodes.keySet().iterator();

    int j;
    while(endPositionIterator.hasNext()) {
        j = (Integer)endPositionIterator.next();
        if ((Boolean)this.visitedNodes.get(j)) {
            Cell node = findCellInGrid(this.grid, j);
            this.drawRectangle(g, node.getX() * this.cellSize, node.getY() * this.cellSize, this.cellSize, Color.cyan);
        }
    }

    int i;
    for(i = 0; i < rows; ++i) {
        for(j = 0; j < columns; ++j) {
            if (this.grid[i][j].isWall()) {
                this.drawRectangle(g, j * this.cellSize, i * this.cellSize, this.cellSize, Color.LIGHT_GRAY);
            }
        }
    }

    this.drawRectangle(g, this.startNode.getX() * this.cellSize, this.startNode.getY() * this.cellSize, this.cellSize, Color.RED);
    endPositionIterator = this.endNodes.iterator();

    while(endPositionIterator.hasNext()) {
        Point p = (Point)endPositionIterator.next();
        this.drawRectangle(g, p.getX() * this.cellSize, p.getY() * this.cellSize, this.cellSize, Color.GREEN);
    }

    g.setColor(Color.BLACK);

    for(i = 1; i < rows; ++i) {
        g.drawLine(x1: 0, i * this.HEIGHT / rows, this.WIDTH, i * this.HEIGHT / rows);
    }

    for(i = 1; i < columns; ++i) {
        g.drawLine(i * this.WIDTH / columns, y1: 0, i * this.WIDTH / columns, this.HEIGHT);
    }

    if (this.path_found) {
        endPositionIterator = this.path.iterator();
```



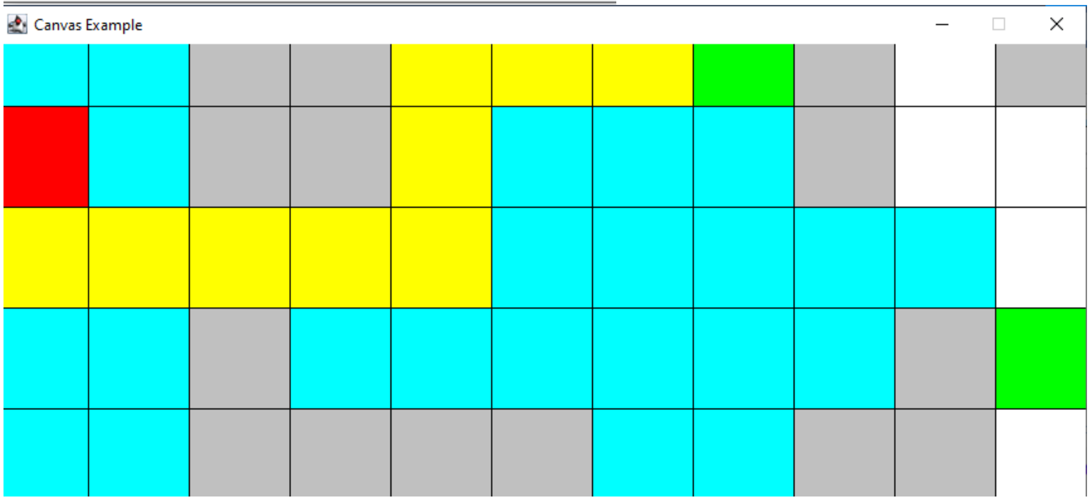
```java
    if (this.path_found) {
        endPositionIterator = this.path.iterator();

        while(endPositionIterator.hasNext()) {
            Cell c = (Cell)endPositionIterator.next();
            this.drawRectangle(g, c.getX() * this.cellSize, c.getY() * this.cellSize, this.cellSize, Color.YELLOW);

            try {
                Thread.sleep(millis: 150L);
            } catch (InterruptedException endPositionIterator8) {
                Logger.getLogger(MyCanvas.class.getName()).log(Level.SEVERE, (String)null, endPositionIterator8);
            }

            this.drawRectangle(g, this.startNode.getX() * this.cellSize, this.startNode.getY() * this.cellSize, this.cellSize, Color.RED);
            Iterator endPositionIterator12 = this.endNodes.iterator();

            while(endPositionIterator12.hasNext()) {
                Point p = (Point)endPositionIterator12.next();
                this.drawRectangle(g, p.getX() * this.cellSize, p.getY() * this.cellSize, this.cellSize, Color.GREEN);
            }

            g.setColor(Color.BLACK);

            for(i = 1; i < rows; ++i) {
                g.drawLine(x1: 0, i * this.HEIGHT / rows, this.WIDTH, i * this.HEIGHT / rows);
            }

            for(i = 1; i < columns; ++i) {
                g.drawLine(i * this.WIDTH / columns, y1: 0, i * this.WIDTH / columns, this.HEIGHT);
            }
        }
    }

}

void drawRectangle(Graphics g, int x, int y, int cellSize, Color c) {
    g.setColor(c);
    g.fillRect(x, y, cellSize, cellSize);
}

static Cell findCellInGrid(Cell[][] grid, int id) {
    Cell result = null;
    boolean found = false;

    for(int i = 0; i < grid.length && !found; ++i) {
        for(int j = 0; j < grid[i].length && !found; ++j) {
            if (grid[i][j].getId() == id) {
                result = grid[i][j];
                found = true;
```



Robot Navigation

WELCOME Select Map map1.txt Select Algorithm bfs

Run


Canvas Example