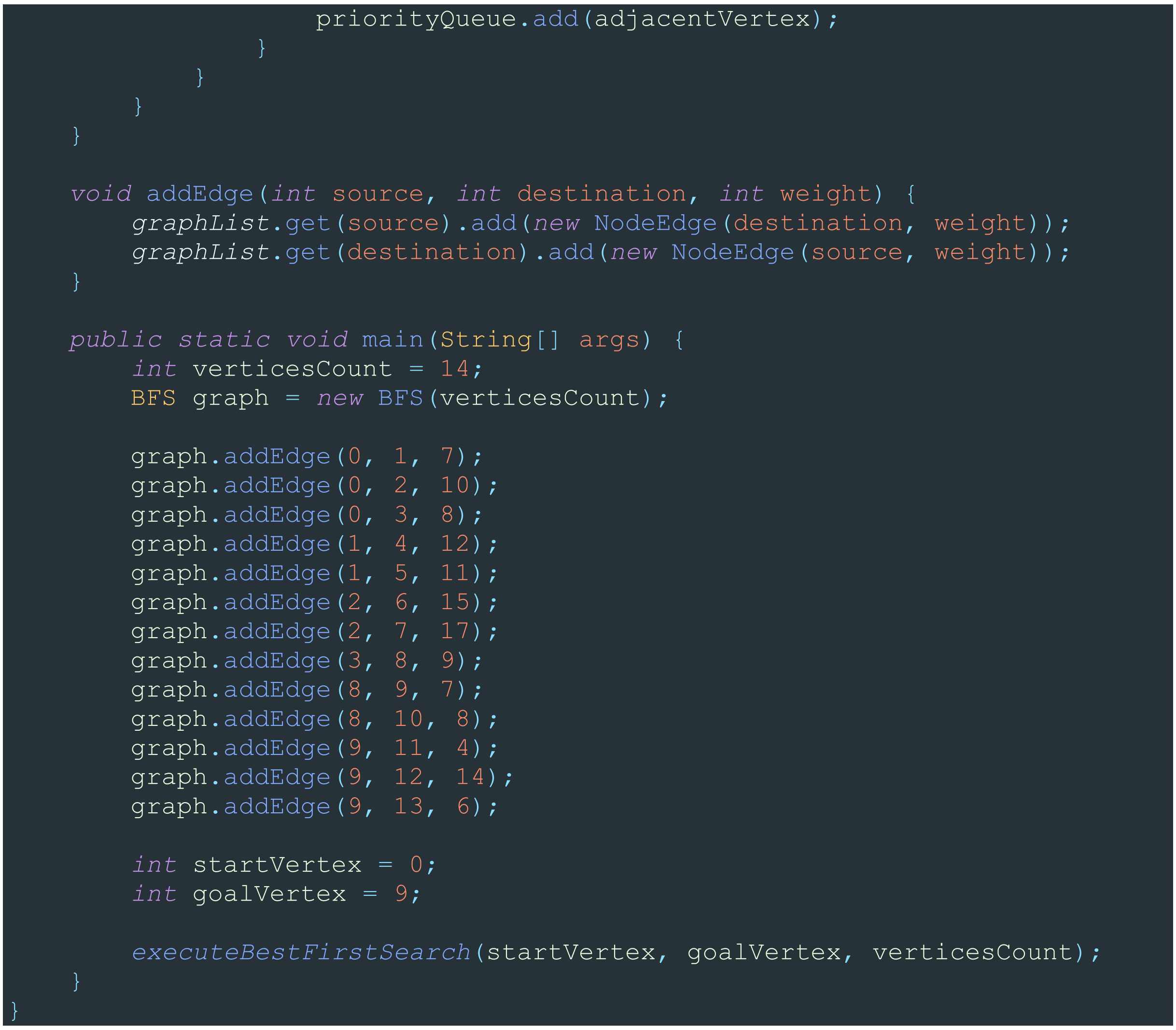
AI Assignment 3 - Implementation of Informed Strategies

Name : Siddhi Rajeshirke
Division : CS-C
Batch : 3
Roll No : 75
PRN : 12110298

1.) BFS(Best First Search)

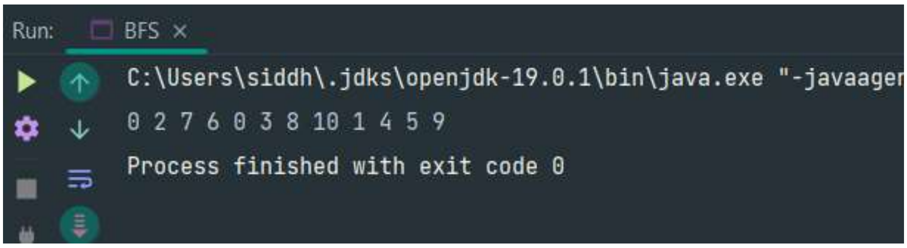
Code :

```
import java.util.*;

public class BFS {

    static ArrayList<ArrayList<NodeEdge>> graphList = new ArrayList<>();

    static class NodeEdge implements Comparable<NodeEdge> {
        int vertex, weight;

        NodeEdge(int vertex, int weight) {
            this.vertex = vertex;
            this.weight = weight;
        }

        @Override
        public int compareTo(NodeEdge otherEdge) {
            return otherEdge.weight - this.weight;
        }
    }

    public BFS(int vertices) {
        for (int i = 0; i < vertices; i++) {
            graphList.add(new ArrayList<>());
        }
    }

    static void executeBestFirstSearch(int startVertex, int goalVertex, int
totalVertices) {
        PriorityQueue<NodeEdge> priorityQueue = new PriorityQueue<>();
        boolean[] isVisited = new boolean[totalVertices];

        priorityQueue.add(new NodeEdge(startVertex, 0));

        while (!priorityQueue.isEmpty()) {
            int currentVertex = priorityQueue.poll().vertex;
            System.out.print(currentVertex + " ");

            if (goalVertex == currentVertex) {
                break;
            }

            for (NodeEdge adjacentVertex : graphList.get(currentVertex)) {
                if (!isVisited[adjacentVertex.vertex]) {
                    isVisited[adjacentVertex.vertex] = true;
```



```
                    priorityQueue.add(adjacentVertex);
                }
            }
        }
    }

    void addEdge(int source, int destination, int weight) {
        graphList.get(source).add(new NodeEdge(destination, weight));
        graphList.get(destination).add(new NodeEdge(source, weight));
    }

    public static void main(String[] args) {
        int verticesCount = 14;
        BFS graph = new BFS(verticesCount);

        graph.addEdge(0, 1, 7);
        graph.addEdge(0, 2, 10);
        graph.addEdge(0, 3, 8);
        graph.addEdge(1, 4, 12);
        graph.addEdge(1, 5, 11);
        graph.addEdge(2, 6, 15);
        graph.addEdge(2, 7, 17);
        graph.addEdge(3, 8, 9);
        graph.addEdge(8, 9, 7);
        graph.addEdge(8, 10, 8);
        graph.addEdge(9, 11, 4);
        graph.addEdge(9, 12, 14);
        graph.addEdge(9, 13, 6);

        int startVertex = 0;
        int goalVertex = 9;

        executeBestFirstSearch(startVertex, goalVertex, verticesCount);
    }
}
```

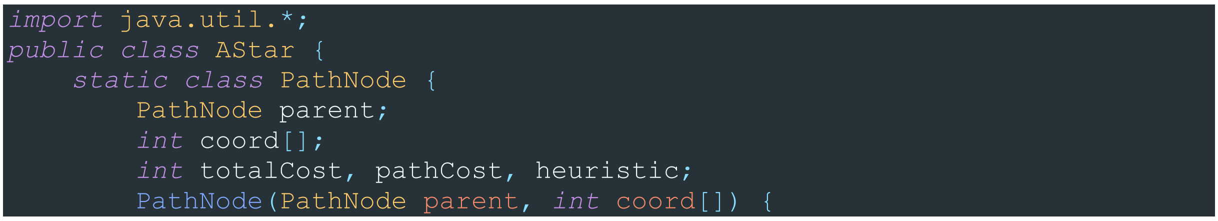
Output :



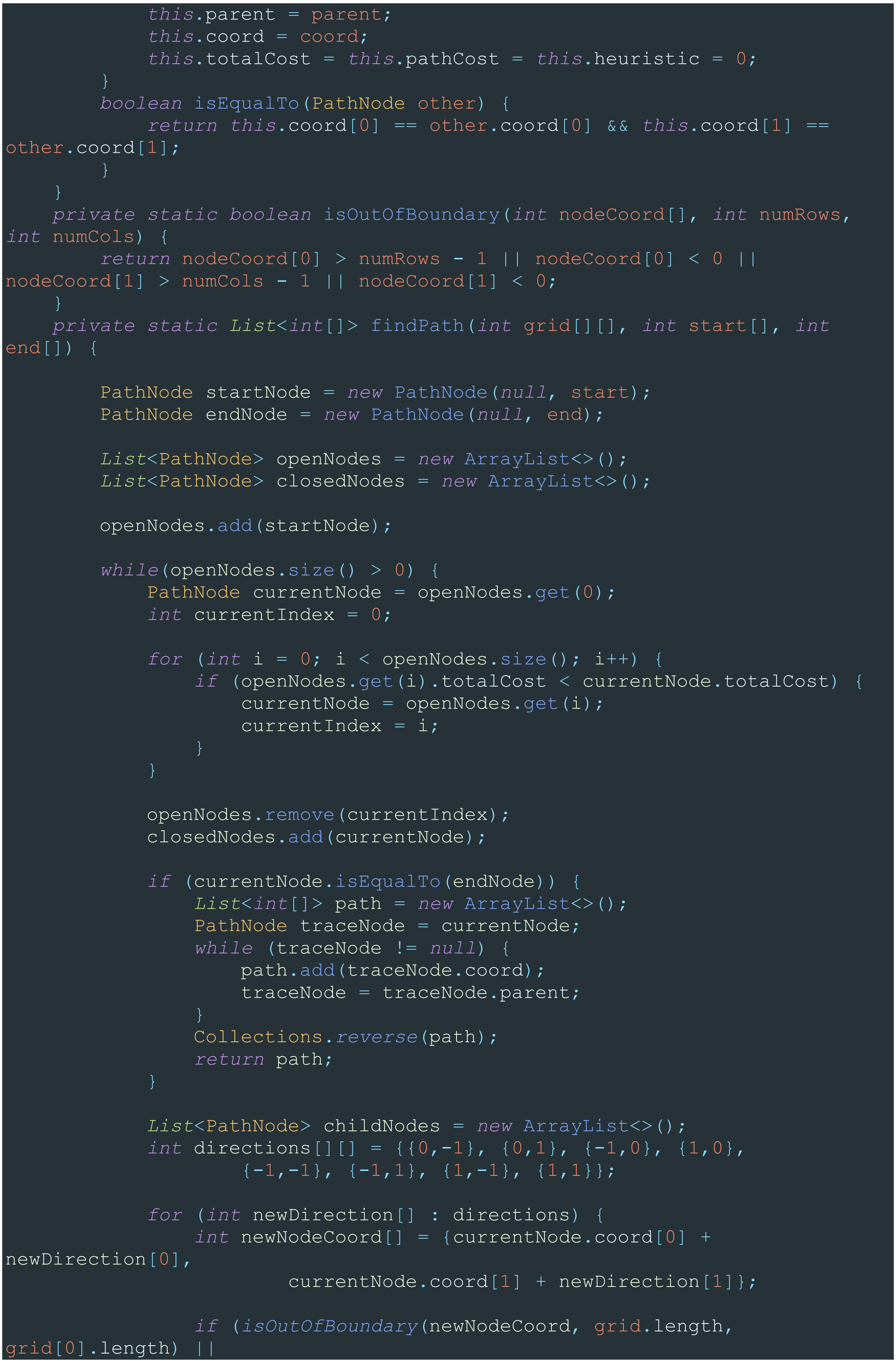
```
C:\Users\siddh\.jdks\openjdk-19.0.1\bin\java.exe "-javaagen
0 2 7 6 0 3 8 10 1 4 5 9
Process finished with exit code 0
```

2.) A*

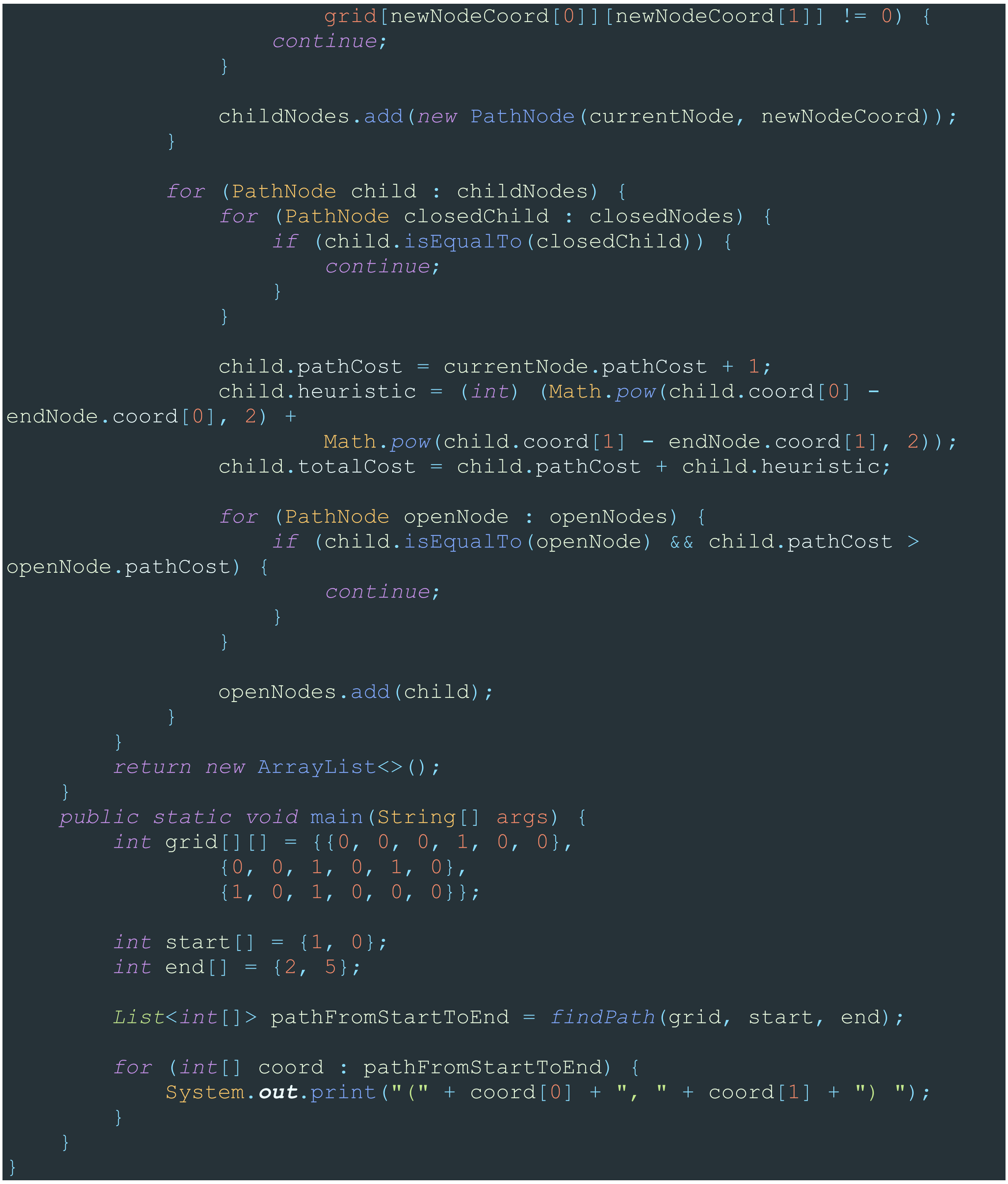
Code :



```
import java.util.*;
public class AStar {
    static class PathNode {
        PathNode parent;
        int coord[];
        int totalCost, pathCost, heuristic;
        PathNode(PathNode parent, int coord[]) {
```

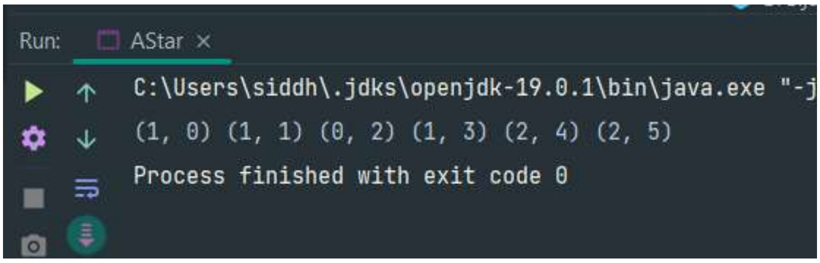


```
            this.parent = parent;
            this.coord = coord;
            this.totalCost = this.pathCost = this.heuristic = 0;
        }
        boolean isEqualTo(PathNode other) {
            return this.coord[0] == other.coord[0] && this.coord[1] ==
other.coord[1];
        }
    }
    private static boolean isOutOfBoundary(int nodeCoord[], int numRows,
int numCols) {
        return nodeCoord[0] > numRows - 1 || nodeCoord[0] < 0 ||
nodeCoord[1] > numCols - 1 || nodeCoord[1] < 0;
    }
    private static List<int[]> findPath(int grid[][], int start[], int
end[]) {

        PathNode startNode = new PathNode(null, start);
        PathNode endNode = new PathNode(null, end);

        List<PathNode> openNodes = new ArrayList<>();
        List<PathNode> closedNodes = new ArrayList<>();

        openNodes.add(startNode);

        while(openNodes.size() > 0) {
            PathNode currentNode = openNodes.get(0);
            int currentIndex = 0;

            for (int i = 0; i < openNodes.size(); i++) {
                if (openNodes.get(i).totalCost < currentNode.totalCost) {
                    currentNode = openNodes.get(i);
                    currentIndex = i;
                }
            }

            openNodes.remove(currentIndex);
            closedNodes.add(currentNode);

            if (currentNode.isEqualTo(endNode)) {
                List<int[]> path = new ArrayList<>();
                PathNode traceNode = currentNode;
                while (traceNode != null) {
                    path.add(traceNode.coord);
                    traceNode = traceNode.parent;
                }
                Collections.reverse(path);
                return path;
            }

            List<PathNode> childNodes = new ArrayList<>();
            int directions[][] = {{0,-1}, {0,1}, {-1,0}, {1,0},
                    {-1,-1}, {-1,1}, {1,-1}, {1,1}};

            for (int newDirection[] : directions) {
                int newNodeCoord[] = {currentNode.coord[0] +
newDirection[0],
                        currentNode.coord[1] + newDirection[1]};

                if (isOutOfBoundary(newNodeCoord, grid.length,
grid[0].length) ||
```



```
                        grid[newNodeCoord[0]][newNodeCoord[1]] != 0) {
                    continue;
                }

                childNodes.add(new PathNode(currentNode, newNodeCoord));
            }

            for (PathNode child : childNodes) {
                for (PathNode closedChild : closedNodes) {
                    if (child.isEqualTo(closedChild)) {
                        continue;
                    }
                }

                child.pathCost = currentNode.pathCost + 1;
                child.heuristic = (int) (Math.pow(child.coord[0] -
endNode.coord[0], 2) +
                        Math.pow(child.coord[1] - endNode.coord[1], 2));
                child.totalCost = child.pathCost + child.heuristic;

                for (PathNode openNode : openNodes) {
                    if (child.isEqualTo(openNode) && child.pathCost >
openNode.pathCost) {
                        continue;
                    }
                }

                openNodes.add(child);
            }
        }
        return new ArrayList<>();
    }
    public static void main(String[] args) {
        int grid[][] = {{0, 0, 0, 1, 0, 0},
                {0, 0, 1, 0, 1, 0},
                {1, 0, 1, 0, 0, 0}};

        int start[] = {1, 0};
        int end[] = {2, 5};

        List<int[]> pathFromStartToEnd = findPath(grid, start, end);

        for (int[] coord : pathFromStartToEnd) {
            System.out.print("(" + coord[0] + ", " + coord[1] + ") ");
        }
    }
}
```

Output :



```
C:\Users\siddh\.jdks\openjdk-19.0.1\bin\java.exe "-j
(1, 0) (1, 1) (0, 2) (1, 3) (2, 4) (2, 5)
Process finished with exit code 0
```