AI-Project-1: Informed Search (Maze)

Student Name: Javid Guliyev

Student ID: g34987427

Introduction:

The following report explains the maze option using informed search algorithms. Option 1 requires determining whether a path exists between points in a maze grid using informed search techniques. The maze is represented as a combination of a 0-1 grid, where '0' denotes an open space and '1' denotes an obstacle.

The language of choice for this task was Java, as the writer of this report is comfortable using strongly-typed languages. The complete source code is available publicly on GitHub via the following link: https://github.com/Cavid2002/AI-Project-1

General Structure and Classes:

There are two files. The first file contains the Main class which gets user input and passes control to the GridSearch class stored in the second file. The GridSearch class contains several variables such as rows, columns, and a reference to an array of 1-0 integers. The purpose of the two static arrays dx and dy is to define directions (up, down, left, right).

```java
public class GridSearch
{
    public int row;
    public int col;
    public int[] map;

    private static final int[] dx = {-1, 1, 0, 0};
    private static final int[] dy = {0, 0, -1, 1};

    GridSearch(String filename, int row, int col)···

    void print_map()···

    public boolean isValid(Point p)···

    public int heuristics(Point p1, Point p2)···

    public int aStart(Point start, Point stop)···

    public int ucs(Point start, Point stop)···

}
```

```java
GridSearch(String filename, int row, int col)
{
    this.row = row;
    this.col = col;
    this.map = new int[row * col];
    int c = 0;
    try
    {
        FileInputStream file = new FileInputStream(filename);
        int b;
        while((b = file.read()) != -1)
        {
            if(b != '1' && b != '0')
            {
                continue;
            }

            map[c++] = b - '0';
        }

        file.close();
    }
    catch(Exception e)
    {
        System.err.println(x:"File IO Error");
        System.exit(status:1);
    }
}
```

By calling the constructor of the GridSearch class (as pictured above) and passing the filename with the grid dimensions, the array will be populated with zeros and ones.

Additionally, there are two helper classes. The first is the **Point** class, which stores the Cartesian coordinates of a point. The second important helper class is the **Node** class, which contains a reference to the point and the cumulative cost from the start to that specific point, combined with heuristics. This class is utilized exclusively by the priority queue to rearrange the nodes based on the lowest cost value.

The contents of both classes are provided in images below:

```java
class Point
{
    public int x;
    public int y;

    Point(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    @Override
    public boolean equals(Object o)
    {
        if (this == o) return true;
        if (!(o instanceof Point)) return false;
        Point point = (Point) o;
        return x == point.x && y == point.y;
    }

    @Override
    public int hashCode()
    {
        return Objects.hash(x, y);
    }
}
```

```java
class Node implements Comparable<Node>
{
    int cost;
    Point p;

    Node(Point p, int cost)
    {
        this.cost = cost;
        this.p = p;
    }

    @Override
    public int compareTo(Node other)
    {
        return Integer.compare(this.cost, other.cost);
    }

}
```

Informed Search:

The informed algorithm of choice for this task was A*. Unlike uninformed search (UCS), where the path with the least cumulative cost is chosen without considering the distance to the goal node, A* combines the estimated cost from the current point to the goal (as in Greedy Search) with the cumulative cost from the start to the current point (as in UCS). A* is essentially a combination of UCS and Greedy Search, or UCS search with heuristics.

```java
public int heuristics(Point p1, Point p2)
{
    int mdist = Math.abs(p1.x - p2.x) + Math.abs(p1.y - p2.y);
    return mdist;
}
```

As observed in the image above, heuristics are calculated by finding the Manhattan distance between two points. The result of this calculation is added to the cumulative cost to obtain the final cost, which is then pushed onto the priority queue to guide further decisions on which direction to expand.

Before explaining the algorithm, it's essential to highlight another method that checks whether a given point is an obstacle or out of bounds:

```java
public boolean isValid(Point p)
{
    if(p.x < 0 || p.y < 0 || p.x >= col || p.y >= row) return false;
    return map[p.y * this.col + p.x] == 0;
}
```

The complete algorithm for A* in grid maze with explanation is provided below:

```java
public int aStar(Point start, Point stop)
{
    PriorityQueue<Node> pq = new PriorityQueue<>();
    HashMap<Point, Integer> gCostMap = new HashMap<>();

    gCostMap.put(start, value:0);
    pq.add(new Node(start, heuristics(start, stop)));

    while(!pq.isEmpty())
    {
        Node current = pq.poll();
        Point currPoint = current.p;
        int gCost = gCostMap.get(currPoint);

        if(currPoint.equals(stop)) return gCost;

        for(int i = 0; i < 4; i++)
        {
            Point np = new Point(currPoint.x + dx[i], currPoint.y + dy[i]);

            if(isValid(np) == false) continue;

            int h = heuristics(np, stop);
            int g = gCost + 1;
            int f = g + h;

            if(!gCostMap.containsKey(np) || gCostMap.get(np) > g)
            {
                gCostMap.put(np, g);
                pq.add(new Node(np, f));
            }

        }

    }

    return -1;
}
```

Priority queue: is responsible for rearranging the Nodes based on the least fCost value which is cumulative cost combined with heuristics: f(n) = g(n) + h(n). This queue would be responsible for providing the cheapest path to expand to.

HashMap: is keeping track the minimal cumulative cost(value) to reach the particular Point(key).

The algorithm steps:

1. Insert start point into priority queue and into hash map;
2. Repeat until queue is empty
    a. Pop the Node from the queue and extract point
    b. Check if the current point is a goal point if yes return cumulative cost to reach that point from hash map; else proceed further
    c. Do for all 4 directions(upward, downward, leftward, rightward): check if it is not an obstacle; calculate heuristics from the point to the goal; insert them into the queue if point is not visited or if cost is less than current cost;
3. Return -1 if path is not found

Results of the Algorithm:

The addition to “YES” and “NO” output results it is also provided the cost from start point to result point. The results and their inputs are provided in the table below:

Start Point	End Point	Output	Cost
1,34	15,47	YES	53
1,2	3,39	YES	51
0,0	3,77	NO	-1
1,75	8,79	YES	11
1,75	39,40	YES	91