# Karel Report

ESSAM ABDO

# Table of Contents



# Table of Figures

# 1 Introduction

For This report I demonstrated my solution for Karel problem , I am going to explain the two methods I chose to solve the problem , and a comparison between the solutions .

# 2 Problem Statement

Divide a given map of any size ( the smallest size is 5 * 5 ) into 4 equal areas then locate the center point ( or square ) .

Optimize your solution as follows :

- Karel should achieve his task with the lowest number of moves. Add a moves counter to your code and print it while Karel is moving.
- You should minimize the number of lines in your code to the lowest possible number of lines by writing reusable functions.

# 3 Greedy Solution

When I read the assignment , immediately I started visualising the map and thinking about the route that Karel is going to walk in . But after several trials , and after a long time of planning , I chose to let Karel choose the way he wants to walk in .

## 3.1 Methodology

Karel was able to decide the path he wants to walk in , using the following methodology :

```
private void solution1()
{
    setBeepersInBag(1000);
    initializeVariables();
    findSize( destination: "up");
    //findSize("right");
    findDesiredPoints();
    while(points.size()!=0)
    {
        calculateCosts();
        goToPoint(points.get(0).getX(),points.get(0).getY());
    }
    goToPoint( x: 1, y: 1);
    System.out.println("steps = " + steps + "\n");
}

public void run()
{
    solution1();
}
```

**Figure 1 Solution 1 Driven Function**

1 -  Firstly , Karel find the size of the , either by heading up or right .

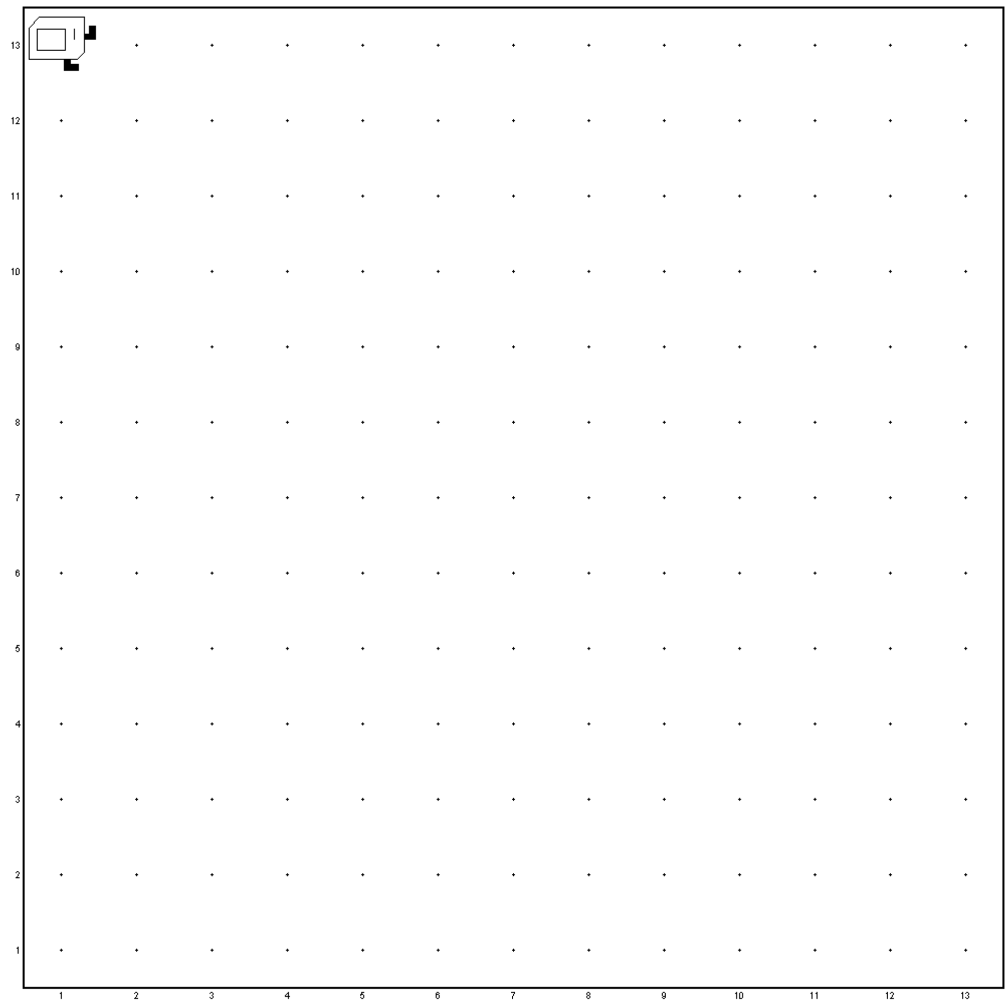

**Figure 2 The Starting point for Karel**

2 - After finding the size , the starting point of my program is going to be ( n , 1 ) , where n is the number of columns and rows , a function called **findDesiredPoints()** will be invoked , this function uses map size to find all the points that we need to place a beeper on , and will append them to an ArrayList of class **Point** called **points** , the Figure below demonstrates in red the points that will be added to **points** list .

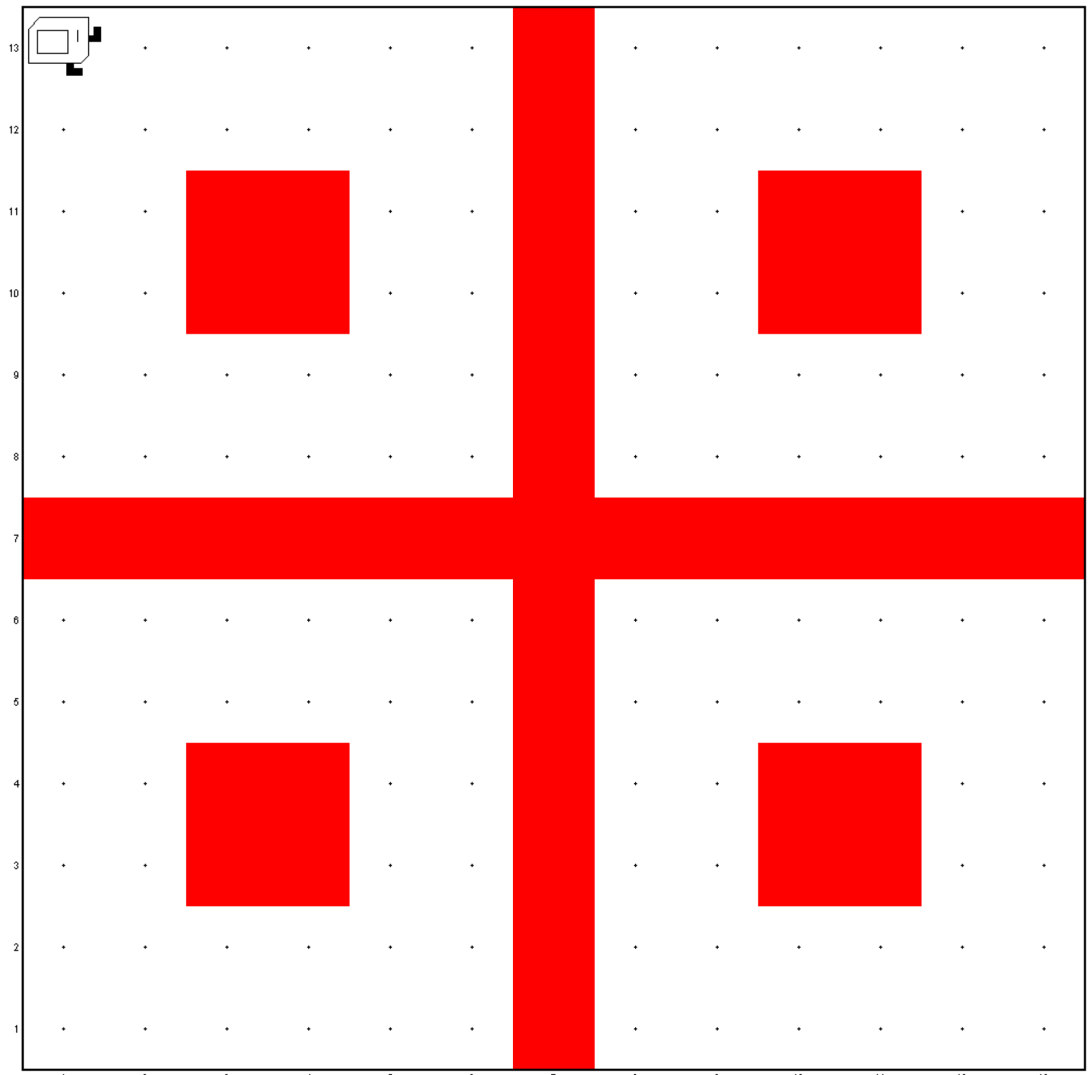

**Figure 3 The Target Points for Karel**

This is a snapshot of Point class :

```
class Point implements Comparable<Point>
{
    private int x;
    private int y;
    private int cost = 0;

    Point(int x,int y)
    {...}

    public int getX() { return x; }

    public int getY() { return y; }

    public int getCost() { return cost; }

    public void calculateCostFrom(int row,int column)
    {
        this.cost = Math.abs(row - this.x) + Math.abs(column - this.y);
    }

    // overriding the compareTo method of Comparable class
    @Override public int compareTo(Point other_point)
    {...}

    @Override
    public boolean equals(Object o)
    {...}

    @Override
    public int hashCode()
    {...}
}
```

**Figure 4 Point Class**

3 - After setting our target points , now we need to go to each one of them and place a beeper , Karel will visit the closest target point to him , in other words , **points** list will be ordered according to the number of steps needed to reach each point from the current location . **calculateCosts()** function will invoke **calculateCostFrom(int row,int column)** , a public method in class **Point** , for each point in **points list** , so we can order the points in the list according to the number of steps needed to reach each one of them .

4 - After sorting the points , now we head to the closest one of Karel's current location , using the function **goToPoint(int x,int y)** , this function will move Karel to the point that is located in ( x , y ) coordinate .

5 - Step 3 and Step 4 are going to be repeated until we visit each point of our targets .

6 - After placing the beepers on all the target points , now Karel will head back to the point ( 1 , 1 ) .

## 3.2 Some Notes

** Steps are count in **myMove()** function , this is a snapshot of it :

```
private void myMove()
{
    if(facingWest())
    {
        current_column--;
    }
    else if(facingEast())
    {
        current_column++;
    }
    else if(facingSouth())
    {
        current_row--;
    }
    else if(facingNorth())
    {
        current_row++;
    }
    move();
    checkIfThisIsATarget();
    steps++;
}
```

**Figure 5 myMove() function**

This function is used to track Karel's current location , move , check if the current point is one of our targets ( so we can get rid of some points along the way while moving ) , and count the number of steps .

# 4 A* Solution

After I finished coding the first solution , I noticed some insufficiency in Karel's performance for some maps ( it does not always walks the optimal number of steps ) , this is because my greedy solution is not optimal . This is why I started thinking about another efficient solution , and A* crossed my mind .

## 4.1 A* Definition

A* is a search algorithm that uses heuristic function as a guide while search . It basically searches for the best solution of a problem guided by an evaluation function that evaluates every state we visit , ordering them in a heap , then examine each one of them in order , if the current state ( which is the best state ) was not a goal state , we expand this node ( its children ) and add them to the heap , then do the same steps again , until a goal state is found .

A* is a graph traversal and path search algorithm, which is often used in many fields of computer science due to its completeness, optimality, and optimal efficiency. One major practical drawback is its $O(b^d)$ space complexity, as it stores all generated nodes in memory . [1]

## 4.2 The way I used A* for solving Karel's problem

I constructed a class called **Path** , it will be the object that will get evaluated by the heuristic function , A* is an optimal searching algorithm , but it has to find the solution in a little time , so Karel can start his journey as fast as possible .

### 4.2.1 Tree Expanding

The tree of my A* is demonstrated in the figure below :

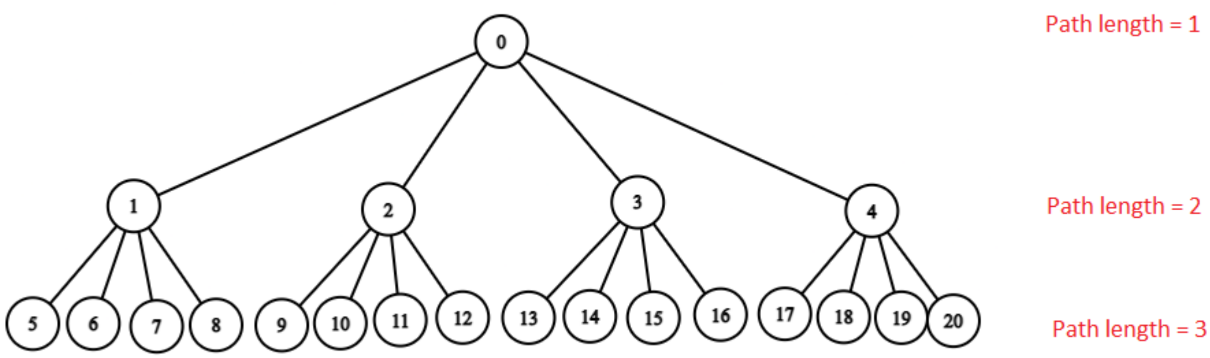


Figure 6 Tree of Paths

I expand the tree and construct different paths like this :

1 - First we append the point ( n , 1 ) to a list of paths .
2 - We expand this path for the 4 closest target points of the last added point to the path .
3 - After we expand four nodes ( each one of them is a duplicate of the parent but has one additional point ) . Then all the children will be appended to a list of paths .
4 - We sort the paths according to the value returned by the heuristic function , then we get the best path ( with the lowest cost ) and expand its nodes and append them to the list of paths .
5 - Step 4 is going to be repeated until we find a goal state ( the lowest possible cost of a path that leads us through the way of visiting the target points then heading to the point (1 , 1 ) ) .

## *Some Notes*

We cannot continue expanding the list with a branching factor of 4 , some trees will reach a depth up to 200 , so I put some constraints on the way I expand my tree . The following figure demonstrates a snapshot of function **getChildren()** in the **Path** class :

```
public ArrayList<Path> getChildren()
{
    int maximum_branching_factor = 4;
    if(this.path_points.size()>6)
    {
        maximum_branching_factor =1;
    }
```

**Figure 7 getChildren() function**

As you can see , I expand the nodes up to 4 children until the depth of the tree exceeds 6 , then we will always expand the node to only one child ( only the closest point to the last added point to the path ) , this is done because we need to find the answer in a fast way . This way maybe will not get the best solution , but at least we get a good one with a reasonable time .

### 4.2.2 The heuristic function

The figure below shows a snapshot of a member function in class **Path** that is used to give an estimation of how good this path is :

```
private void updateAllCost()
{
    this.path_cost=0;
    double h1 = 0;
    for (int i = 0; i < this.path_points.size() -1; i++)
    {
        h1 += findCostBetweenTwoPoints(this.path_points.get(i),this.path_points.get(i+1));
    }
    h1/=398;
    h1*=100;
    double h2 = 1 - this.must_visit.size()/desiredPointsSize;
    h2*=100;
    this.path_cost = (h1 + h2);
}
```

**Figure 8 The heuristic Function**

The function calculates the estimation using these two values :

1 - The cost of the path , it is converted to a scale from 0 to 100 , I chose 398 as a maximum value because this is the highest number of steps I got for the 50x50 map using the greedy solution .
2 - The number of the target points the path has covered until now , divided by the whole number target points that should be covered , this value is converted to a scale from 0 to 100 .

The total cost is the summation of the first value and (100 – the second value ) . A path with high summation is considered a bad path , and a path with low summation is considered a good one .

## 4.2.3 Methodology

This is a snapshot of the driven function of my A* solution :

```
private void solution2()
{
    setBeepersInBag(1000);
    initializeVariables();
    findSize( destination: "up");
    //findSize("right");
    findDesiredPoints();
    Path chosen_path = getBestPath();
    ArrayList<Point> chosen_road = chosen_path.getPath_points();
    points=chosen_road;
    points.remove( index: 0);
    points.remove( index: points.size()-1);
    while (points.size()!=0)
    {
        goToPoint(points.get(0).getX(),points.get(0).getY());
    }
    goToPoint( x: 1, y: 1);
    System.out.println("steps = "  + steps);
    System.out.println("Estimated Cost = " + chosen_path.getPath_cost() + "\n");
}

public void run()
{
    solution2();
}
```

**Figure 9 Solution 2 Driven Function**

The solution is demonstrated in the following steps :

1 - Finding the size of the map while heading to the point ( n , 1 ) , where n is the size of the map .
2 - Finding the target points ( the points that should have a beeper on it ) using the size of the map .
3 - Calculating the best possible path in the allowed time using A* .
4 - Visiting all the points of the path in the given order .
5 - Heading back to point ( 1 , 1 ) .

# 5 Greedy Solution VS A* Solution

The greedy solution gave a very good results for a simple solution , it was easier to implement and easier to track and understand , while the other solution ( A* ) was the opposite , it gave a way better results , actually some of Karel's moves were surprising to me , I did not expect some intelligent move from Karel ! , but in the other hand it was hard to implement , track and debug this solution .

Both of the solutions were written in the object-oriented method , every functionality is implemented as a method , so the code can be readable and not long .

The following chart demonstrates a comparison between both Greedy Solution and A* Solution for the number of steps per some maps with different sizes :

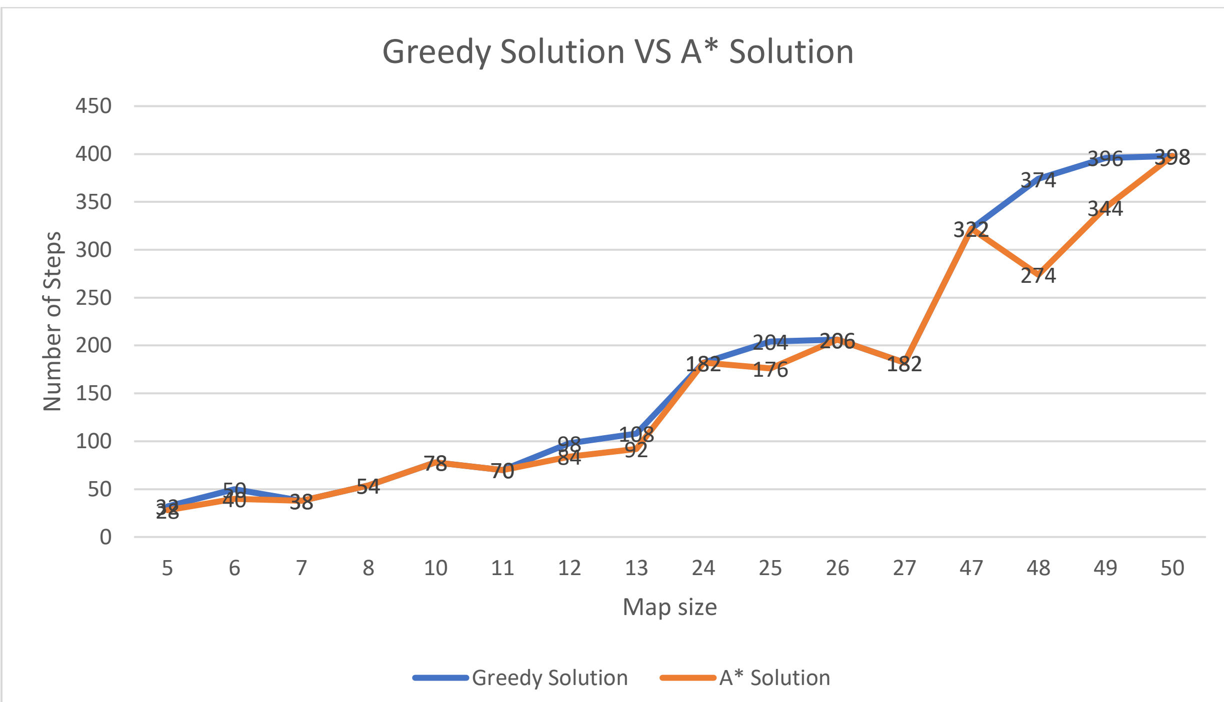


**Figure 10 A Comparison Between Greedy solution and A* for the number of steps per some maps with different sizes**

You can notice that A* Solution beats the greedy solution in multiple tests and in others it meets . while in time complexity , without a doubt , the greedy solution exceeds .

I have submitted the two solution , each one of them in a separate file , and one file has both of them combined ( you can choose which one use before compiling the project ) .

```
public void run()
{
    //solution1();
    solution2();
}
```

Figure 11 run() function

# References


1. *A* search algorithm From Wikipedia*. Available from: https://en.wikipedia.org/wiki/A*_search_algorithm.