CSCI630 – Lab 1 Report

Summer Orienteering

Author : Archit Satish Joshi (aj6082)

Program Execution:

Program is written in python 3.11 and can be run by the command below –

>python lab1.py terrain.png mpp.txt red.txt redOut.png

Arguments that need to be provided are –

1. Terrain image in png format adhering to the legend of colors given in the question.
2. mpp.txt file which has 500*395 elevations for each data point.
3. red.txt is the file with all the control points that NEED to be visited.
4. redOut.png is the name of the output file that the result will be drawn to on top of the terrain image.

Terrain image and mpp.txt need to be 500*395 in dimension or the extra pixels will be stripped in the program.

Assumptions:

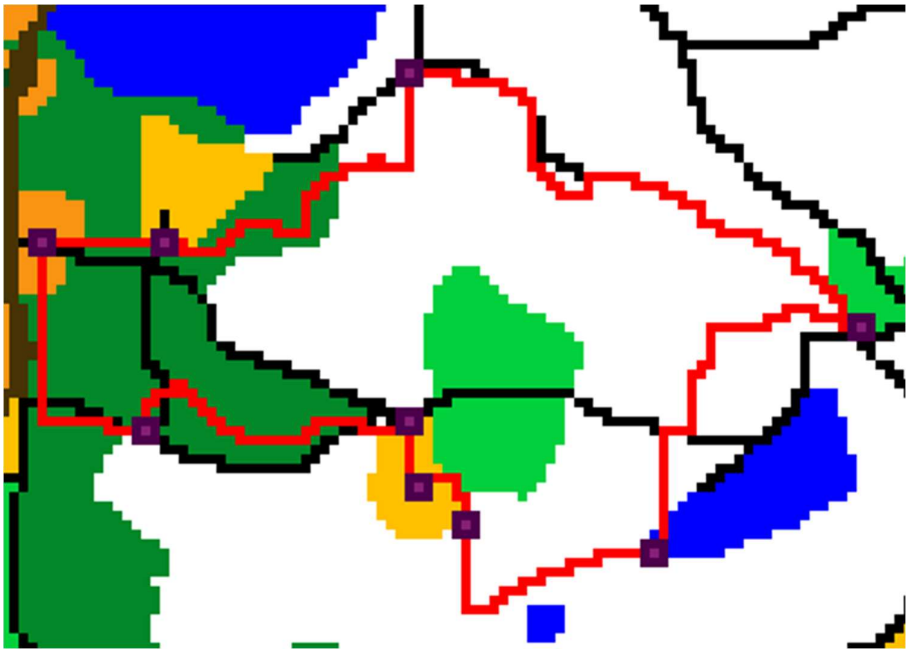
Speeds for different terrain types

```
"""
Terrain Speed Map on a scale of 0-5 (Slowest - Fastest)
{'TerrainType' : Speed}
"""
terrainSpeed = {
    "OpenLand"            : 5,
    "RoughMeadow"         : 1,
    "EasyMoveForest"      : 2,
    "SlowRunForest"       : 2.5,
    "WalkForest"          : 3,
    "ImpassibleVegetation": 0.5,
    "LakeSwampMarsh"      : 0.5,
    "PavedRoad"           : 5,
    "Footpath"            : 4,
    "OutOfBounds"         : 0
}
```

Terrain speed are stored in a map with 5 being the fastest you can walk on a pixel to 0 being the lowest on out of bound regions which are inaccessible. Lake/Swamp/Marsh and Impassible Vegetations are assumed to be crossable very slowly at a speed factor of 0.5

Expected Output:

The path traversed will be drawn in red (RGB (255,0,0)) and the control points will be drawn in light purple (RGB (135, 31, 120)). Additionally, to highlight the control points on the path, the linear and diagonal neighbors of the control points will be drawn with a deep purple color (RGB (77, 0, 75). An example of the output with white.txt as input is given below –



General Idea:

Each pixel on the image, their color values and elevation data will be used to construct 500*395 graph nodes which will be then used in the A* Search algorithm.

Heuristic function (h(n)):

The heuristic value for each node is calculated as the time taken to reach the destination node from that node using the formula -> **time = distance/speed**.

The distance between 2 nodes is calculated by using the Euclidian distance formula in 3D space. We have the x and y from the terrain image. The z coordinate will be deduced from the elevation value for that particular node.

The formula for three-dimension distance is:

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

Where:

- d: the distance between the two points (the hypotenuse)
- x1, y1, z1: the x, y, and z coordinates of point 1
- x2, y2, z2: the x, y, and z coordinates of point 2

Assumption: As this is a heuristic function, we assume that there are no obstacles between the node and destination, hence the movement speed is uniform. In our case the assumption becomes that each pixel is of type “OpenLand” and has a speed value of 5.

Cost between 2 nodes (g(n)):

Distance between 2 nodes is calculated depending on which axis the movement will happen using the longitude and latitude value provided. If the x coordinates of the pixels match, we calculate using the longitude and the Euclidian formula, if the y coordinates of the pixels match, we calculate using the latitude and the Euclidian formula.

```
# Travel along y-axis upwards or downwards / travel along longitude
dist = math.sqrt((longitude ** 2) + (child.elevation - current.elevation) ** 2)
```

```
# Travel along x-axis left or right / travel along latitude
dist = math.sqrt((latitude ** 2) + (child.elevation - current.elevation) ** 2)
```

The final cost (or time) to travel between two nodes here depends on the speed with which we can travel along a node. This mobility will again be decided depending on which axis the travel happens.

If travel happens along positive x-axis we consider mobility for the current pixel itself , its neighbor otherwise along the negative x-axis. If travel happens along positive y-axis we consider the mobility for its neighbor, its own mobility otherwise for travel along negative y-axis.

The mobility value for each node is assigned during node creation using the Terrain speed HashMap as per our assumptions above.

Final Cost (f(n)):

The final cost for each node is the sum of the heuristic and cost between 2 nodes.

$f(n) = g(n) + h(n)$

Neighbors/Children for each node:

During A* search we need the neighbors for each node which is being considered currently.

A neighbor for each node/pixel is defined as the list of the 4 nodes around it that share the same edge as the current node. We also need to handle the extreme cases where –

1. The current node is in a corner. Thus, has only 2 neighbors. The pixel adjacent to it and below it.
2. The current node is along the edges. Thus, has only 3 neighbors. The pixel adjacent to it, the pixel above it and the pixel below it.

A* Search Algorithm

The below pseudocode has been followed for the algorithm. The heuristic (H-Score) and the distance between 2 nodes (G-Score) have already been discussed above.

```
1.   function AStarSearch(Graph, start, target):
2.      calculate the heuristic value h(v) of starting node
3.      add the node to the opened list
4.      while True:
5.         if opened is empty:
6.            break # No solution found
7.         selected_node = remove from opened list, the node with
8.                         the minimum heuristic value
9.         if selected_node == target:
10.           calculate path
11.           return path
12.        add selected_node to closed list
13.        new_nodes = get the children of selected_node
14.        if the selected node has children:
15.           for each child in children:
16.              calculate the heuristic value of child
17.              if child not in closed and opened lists:
18.                 child.parent = selected_node
19.                 add the child to opened list
20.              else if child in opened list:
21.                 if the heuristic values of child is lower than
22.                  the corresponding node in opened list:
23.                    child.parent = selected_node
24.                    add the child to opened list
```