Foundation of Intelligent Systems
Lab 1

Name: Kriti Sharda

Basic Structure for Orienteering Java Project

I have structured and divided the whole project into different classes that makes the project more presentable, readable, reusable and manageable. Following is the structure of my project:

- **AStarTraversal Class:**

  This class performs the AStar Algorithm and contains function to calculate the sucessors, neighbors, travel costs and also updates the map final path that the AStar has suggested.

- **AStarTraversalInfo Class:**

  This class maintains the state the visited list and the final cost that is being updated and returned by the AStar Class.

- **CustomSpeedInformation Class** – implements SpeedInformation Interface:

  This class is used when the user wants to enter some custom weights corresponding to the different terrains in the map. The user enters the file path in the main and enters the weights in the given sequence.

- **DefaultSpeedInformation Class** – implements SpeedInformation Interface:

  This class contains the default weights for the terrain that I have set up according to my ability and understanding.

- **Orienteering Class (Main Class):**

  This is the main driving class for this project. This class sets all the file paths in the beginning. Starts the Orienteering process and once the orienteering is done, creates a new path image and saves it in the current folder.
  The Orienteering method reads the input file and initializes all the pixels and the map of pixels. Finally, It reads the file that contains the Events and the locations and calls their corresponding function.

- **<u>OrienteeringMapInfo Class:</u>**

  This is a class representing the input map that contains information about all the pixels on the map. Each pixel is an instance of the pixel class. The OrienteeringMapInfo class also has methods to fetch all the characteristics of each pixel and also saves the final Path image into the current directory.

- **<u>Pixel Class:</u>**

  This class represents a pixel in the map and has attributes like x-coordinate, y-coordinate, terrain information, elevation information and methods to fetch these values. It also contains HashMaps that help in mapping the terrain color with the terrain type and vice-versa, which helps in fetching the map image and printing back the final map image.

- **<u>S_TraversalPath Class:</u>**

  This class creates a basic structure that is used by the Score-O algorithm. This class has 3 variables i.e.
    - The path taken, which is a list of pixels that has been traversed
    - The Children, which is a list of the traversal paths that have been generated by the current path taken(or set of pixels)
    - The time taken, which stores the total time that has been spent to travel on the current path taken.

  It also has methods to fetch the path taken and the children paths created.

- **<u>TravelNode Class:</u>**

  This class creates a basic structure used by the AStarTraversal class. It contains variables to store the start pixel, end pixel, current pixel, previous pixel, the g (actual path cost) and the h (heuristic path cost) value for the given set of start, end and current pixels.

Implementation of Input files

There are various file inputs for the code. All of the file paths have been manually entered in the beginning of the main method itself. There is no user input presently.

- A function named InitializeMapInfo is used to initialize a Map for the Orienteering class and this map contains all the Pixels, which are basically instances of the Pixel class. The initialize methods take in the file arguments for the elevation file and the color map image file. It reads both the files and initialized all the pixels and sets their x-coordinate, y-coordinate, elevation and terrain type.

- The path to the input file that contains the event type followed by time and the x-y pixel locations that are to be visited is sent as an argument to the function named StartOrienteering, which reads these files and runs the function corresponding to the event type ("Classic" or "Score-O").

Implementation of A* Algorithm

- To implement the A* Algorithm, I used a Custom priority queue which implements the Priority Queue provided by java, that would keep a track of the most likely neighbor pixel value, which has the minimum total cost (f = g(actual path cost) + h(heuristic path cost)) and use that to go further and find an optimal path.
- The successor function from my AStar implementation makes the use of heuristic function to filter the neighboring pixels that and to assign the actual path cost(g) and the heuristic path cost(h) for each set of start, neighbor and target pixels.

- Heuristic function:
  The function named calculateTravelCost is the heuristic function that calculates and returns the heuristic path cost (h). It also calculates the actual path cost(g) when given corresponding parameters as input.

  My heuristic function uses the elevation and terrain type to estimate the time taken for a person to travel in that particular terrain, with the given elevation. Given two pixels. Firstly, I am calculating the Euclidean distance between the given two pixels. I know how far apart they are in the x-y plane but I still need to find the exact distance between those two pixels.

  Next, I used the Elevation Map to find the difference in elevation between the two pixels. Using this elevation and the x-y distance between the points I can now apply the Pythagoras theorem and find the distance along the elevation or the slope between both the pixels. This distance is the exact distance between the two pixels. I am assigning different speed by which a person can travel in different terrains. This is based on my estimation of how fast a person can travel in a given terrain. Now, I am diving this distance by the time in order to get time estimation, i.e. how long a person will take to travel from that pixel to another.

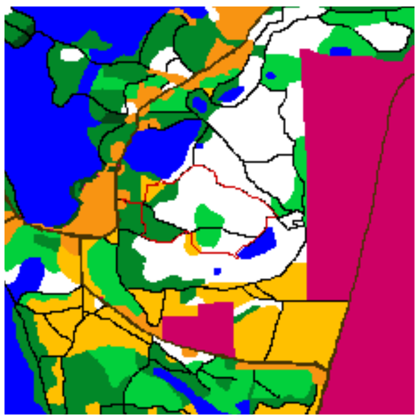
There is still a need to take into consideration the slope or the change in elevation in order to consider the fact that a person may go slower uphill and even slower up a steeper hill. To tackle this, I am calculating the change in elevation by dividing the elevation of target pixel by the elevation of starting pixel and multiplying this change to the time taken to travel in straight path.

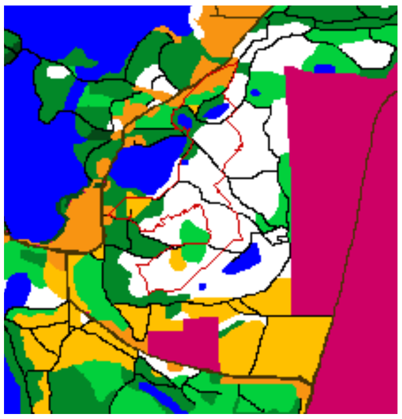
This is the basic idea behind my heuristic function and it works pretty well for all the cases. The successor function uses this to set the g and h values.

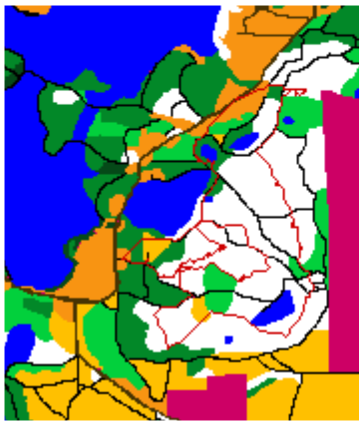
Classic Event Planning

- In this, I am just reading the input classic file and calling the A* algorithm for each pair of consecutive x-y pixel locations mentioned in the file. After the locations have been traversed, the path is printed on the map and a new map is saved as path.png in the same folder that contains the code.

  Final Output Paths:


white path


brown path

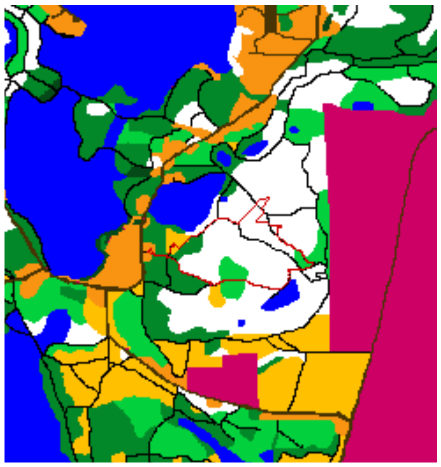

red path

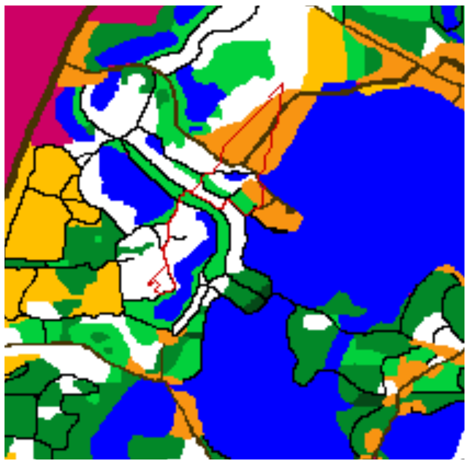
- To make this algorithm work efficiently, I have used concepts from Dynamic Programming while implementing it. The basic idea behind this algorithm is to start with the given start node and store all the possible paths that can be taken to travel with out any restriction of the end location, currently.

- The java class 'S_TraversalPath' defines a structure to store the current path that is a list of locations that were traversed. Moreover, It keeps track of the total cost or time taken to travel on this path and all the children paths of this same class type, that have been generated using this path.

- Finally, as the new path Instances of 'S_TraversedPath' are generated, they are added to a main List named 'scoreList', that keeps track of all the possible paths that can be taken from the start node. The magic of dynamic programming happens here! We are not calculating the total time taken for each path again and again. We are fetching the time taken to travel by the parent path (which is an instance of 'S_TraversalPath') and just adding the extra time taken when a new location is added. This makes out algorithm more efficient.

- Finally, we are traversing the main scoreList and checking for all the paths that have the ending node same as the start node. We keep track of the path, which has visited the most number of locations, and finally print that path and save the final path of the image.


eastesker


allpark

Planning for Someone Else

For this functionality, the user will provide a file that contains a different set of weights. This file is set manually in the main method itself.

There were cases where when we assign equal travel costs to open land, paved roads and footpath the paths changed dramatically as compared to the paths that were generated when open land and paved roads were assigned more weights as compared to the open land.

Every person has a different ability to travel in different terrains and also prefer one terrain to the other. Hence implementing this functionality is very useful.

Some Problems Faced

- Some pixels in the original color terrain map were missing color information and created some exceptions in my code. I had solved this problem by assigning those pixels the terrain value of 'outOfBound'.

- The creation and traversal of the visited list that was implemented using a HashTable in java was a little tricky because the HashTable contained different class instances as its key value pairs and not simple datatypes.

- A problem that I realized a bit later after starting the project was that the image x-y coordinates were not matching with the pixel x-y coordinates that I had implemented. To fix that, I flipped x-y to y-x while creating the pixel instances.

- The inbuilt priority queue of java could not handle a lot of entries, so I had to create a custom priority queue class and make sure that the elements in the list did not exceed a certain threshold.

- Finally, managing the huge code in java was a bit of a task so I had to break down all my code into a lot of classes and methods which took a lot of time but in the end fixed a lot of problems that I was facing.