Rohit Ravishankar
rr9105@rit.edu

Lab 1

CSCI - 630 Foundation of Intelligent Systems

Collaborators: None

INTRODUCTION

According to Wikipedia, **orienteering** is a group of sports that requires navigational skills using a map and compass to navigate from point to point in diverse and usually unfamiliar terrain whilst moving at speed. Participants are given a topographical map, usually a specially prepared orienteering map, which they use to find control points.

This assignment was the first time I heard of orienteering as a sport and hence, it took me some time to get up to speed on how does it work. The links on the assignment page were very helpful in understanding the sport and hence the expectation of the problem.

The problem environment can be defined as:-

State Space: Pixel map with a series of (x, y) values, Elevation & Terrain Map

Initial State: Starting point in (x, y) value for the route

Goal State: Last point in (x, y) value for the route

Actions: Moving to the neighboring pixel (incrementing & decrementing the x & y values)

In order to run the code, we need to be in the root folder of the project and use the following command `python3 runner.py`.

The directory structure for the assignment is divided which in the logical manner to separate the input files from the output files. The root folder contains a directory, **static**, that contains the input files including the terrain map, elevation matrix, the “Classic” event files and the “ScoreO” event file. We create an output folder to store the output of the map with routes marked on them.

The file structure for the assignment is divided based on the functionality each of the file provides. The files and the tasks performed :-

1. runner.py - To run the assignment. The execution of the program starts from here
2. file_imports.py - To perform import of files such as the elevation matrix, the terrain map, files containing the classic courses and the file with the score-o course.

3. utils.py - Provides utility methods for the `runner.py` file including, removing last 5 columns of the elevation matrix and congregating the files to be used by `runner.py`. The file names are hardcoded so, if we want to modify the controls, it should be done within the files contained in the static folder.
4. route_finder.py - The main file for the program which contains the logic for the A * search algorithm
5. route_finder_utils.py - Provides utility functions for the route_finder.py file, such as, finding all the neighbors, calculating the cost of moving to each of the neighbors and the heuristic function

CLASSIC EVENTS

path_A_star_search() - This function implements the A* algorithm and returns the best cost/time to the goal from the given start state, and the parents array (we iterate over the parents array to get the actual path to traverse through the controls in terms of pixels). The function is used in computing the route for the "Classic" event and the "Score-O" event. The function follows the pseudocode for A* search as described in the prescribed textbook. *is_goal()* compares the (x, y) values of the current state to the (x, y) values of the goal state and when *is_goal()* returns true it marks the end of the search process. The A* search algorithm uses a priority queue sorted by a cost value produced by the cost function for each successor of the current state.

cost_function() - The cost function is an important part of the *path_A_star_search()* function. The function uses a few metrics in combination including the terrain, elevation map, direction of movement to calculate the optimal time from the given state to the goal.

I first considered the direction of movement and the terrain data to calculate the distance from one coordinate/pixel to another. Subsequently, I used the pixel data and passed it through a dictionary to find what sort of terrain it is and hence calculated the speed for moving on that terrain. (This is probably clearer in the code section I have written)

I ranked the speeds based on my expectation of the average movement speed on a particular terrain. My choice of speeds for the different terrains were as follows :-

Terrain	**Speed (in m/s)**
Paved Road	4.5
Footpath	3.5
Open Land	3.5
Easy Movement Forest	3
Rough Meadow	2.5
Slow Run Forest	2.5
Walk Forest	1.5
Impassable Vegetation, Lake/Swamp/Marsh, Out Of Bounds	0

To translate this “movement speed” to time, I made use of Jack Daniel’s formula (https://www.livestrong.com/article/533573-running-up-hills-vs-flat-time/), since it seemed to make most sense. The formula states that, “for every percent of incline you experience in an uphill, your running time will slow by 12 to 15 seconds per mile” and “downhills will help you improve your time by approximately eight seconds per mile for every percent gradient of decline”. Based on the gradient, I multiplied the gradient with the distance and multiplied a constant based on whether one is traveling uphill or downhill. I estimated the constant as approximately ((uphill gain/downhill loss) * (Latitude/Longitude) / number of meters in a mile) which was approximately 0.009 and 0.005.

To make a total estimation of the cost/time, I summed up the cost of moving to the next pixel and the cost that is got from the heuristic function.

get_heuristic_distance() - Typically, a good heuristic function is one which has a cost lesser than the actual cost between the current node and successor node. Taking this into account, the Euclidian distance is the shortest distance between any 2 given points and seemed like the best option. However, on further reading, I found that diagonal distances(L_∞) performs better when a 8 direction movement is involved and hence picked diagonal distances as my choice of heuristic function. It was based on my reading from a Stanford theory blog(http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#diagonal-distance). The heuristic function needs you to pick 2 values D and D_2, D being the shortest distance between any 2 points (Longitude/4.5) and D_2 being the diagonal distance ($\sqrt{(Longitude^2 + Latitude^2)}$/4.5). We divide by the fastest speed, i.e., on a paved road, to calculate the heuristic in terms of time. This works fine on flat or level ground but possibly isn’t the best choice for taking elevation into account but for ease of understanding I left out accounting for elevation constants.

classic_route() - This is the calling function to calculate the time spent and plot the route to traverse the classic events of white, red and brown routes.

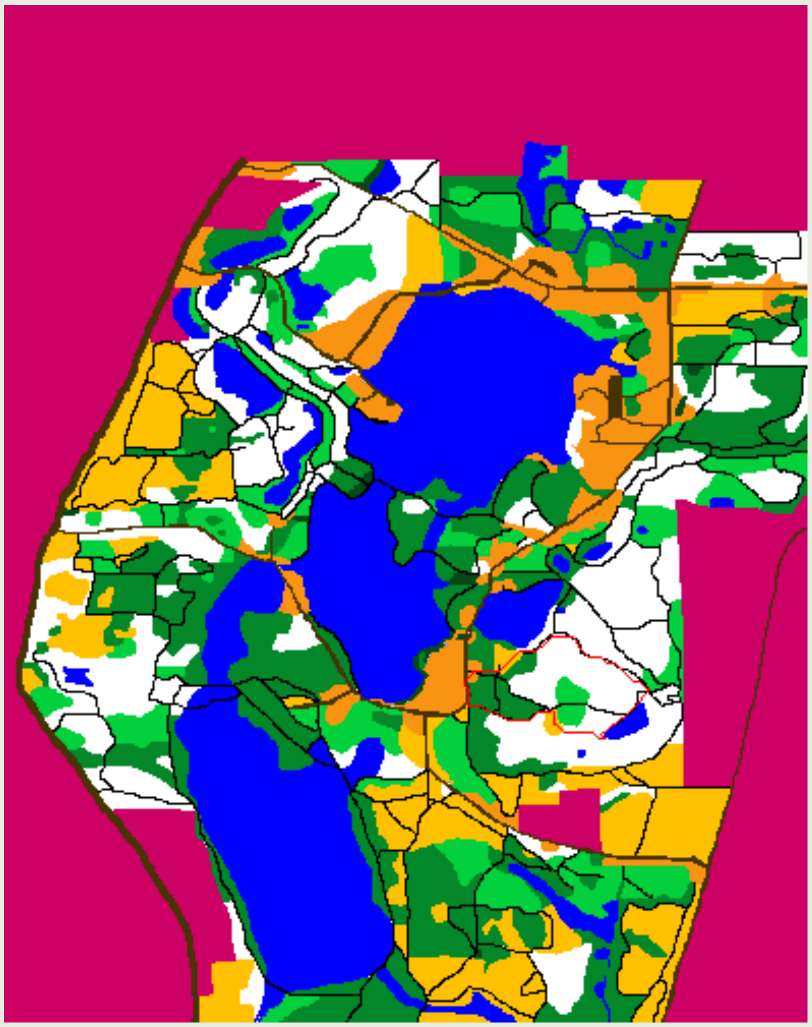
successors() - This function finds all the successors/neighbors for any given pixel. In our case this refers to the 8 adjoining pixels to any given pixel.

draw_path_on_map() - This function plots the path on the terrain map. The parents dictionary that is generated by the *path_A_star_search()* is recursively traversed to find the routing.

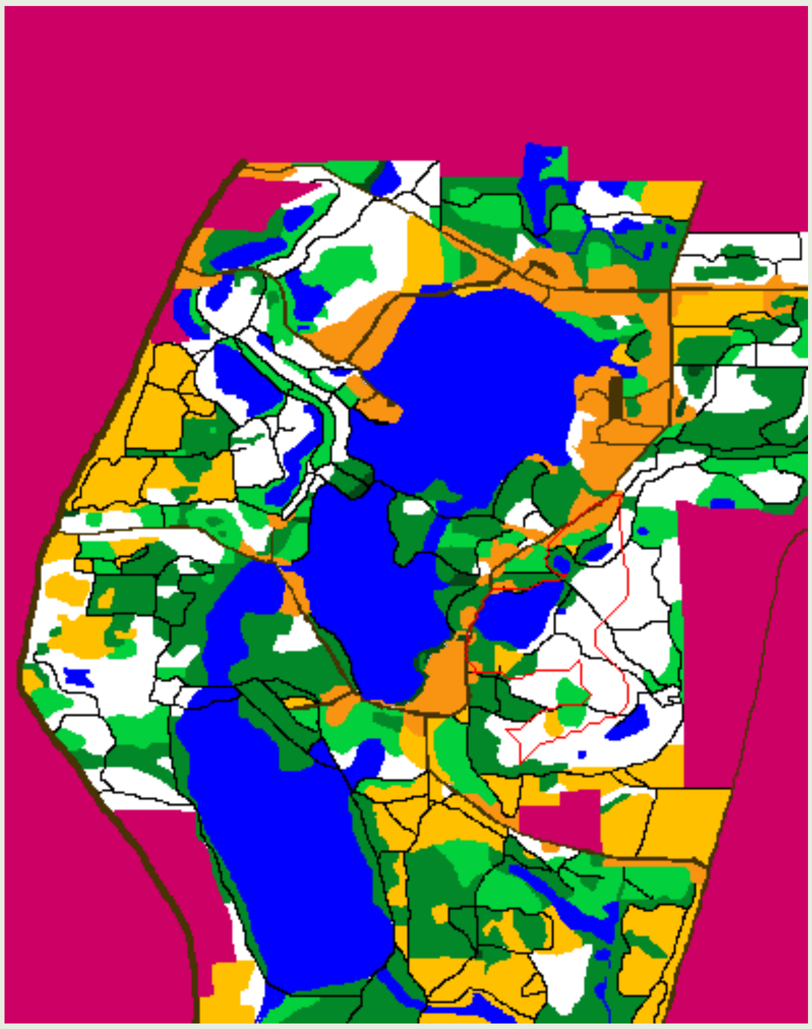
is_goal() - This function finds if the goal state is reached within the A* search algorithm.

The routes received for the classic event are listed below with the number of seconds taken to traverse the path.

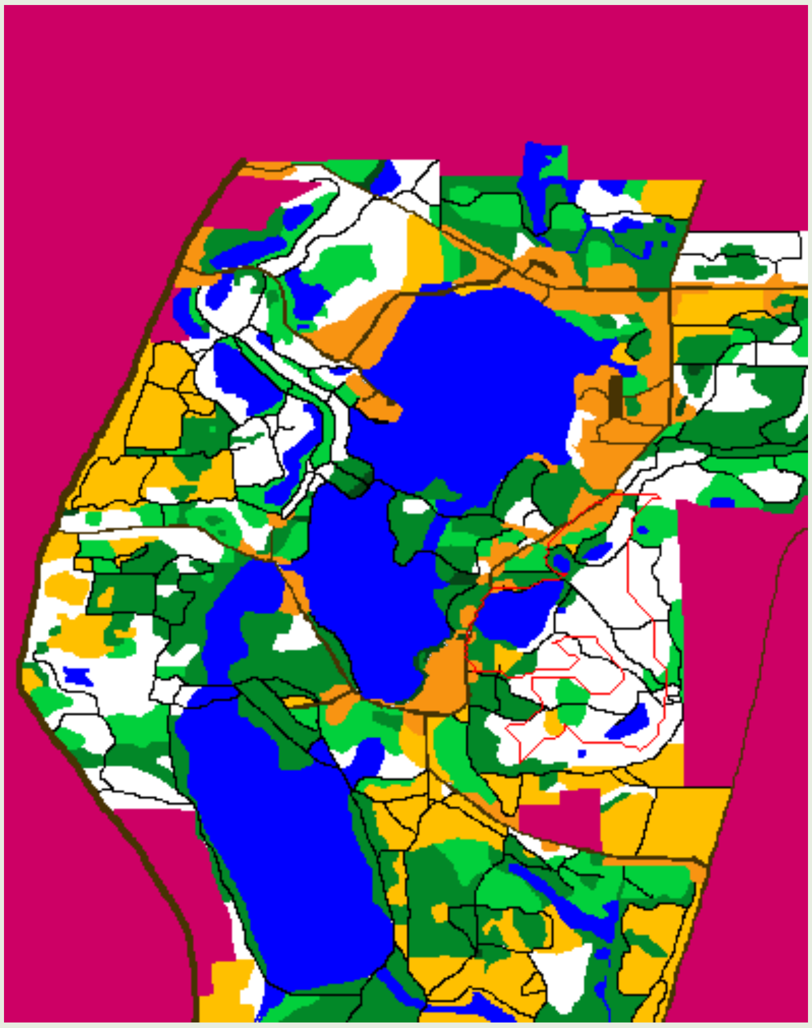
Classic Event - White Route - 763s



Classic Event - Brown Route - 1543s



Classic Event - Red Route - 2041s



SCORE-O EVENT

In order to plan my solution to the problem better, I managed to list down a few additional constraints compared to the classic event straight off the question, i.e., the time to traverse the entire route being limited and the start and end location for the route having to be the same.

The start and end location having to be the same meant that we had find all the permutations for the controls other than the start and finish location. A few references I found online for similar problems such as printing all paths from a given source to destination (https://www.geeksforgeeks.org/find-paths-given-source-destination/), did not help much since I had to traverse the permuted order of the controls between the start and end position, and the "graph" keeps changing.

Another challenge, in my opinion, was deciding how to traverse the adjacency list. The idea to create an adjacency list stemmed from creating an adjacency matrix for the routing algorithm problems such as Dijkstra's algorithm.

A brief description of how my ScoreO algorithm works:-

Akin to how the other files are read, the scoreo file is read and I return the time limit, start/finish location and the list of all controls.

I performed a pre-processing step by creating an adjacency list representation, by means of a dictionary, from all the controls to all the other controls. The list was structured such that each control is a vertex and using the A* search that we previously defined, we find the shortest path and the minimum cost to the other controls. This ensures that the same cost isn't calculated multiple times. This is the **first** optimization I implemented.

Armed with the adjacency list, I fixed the start and end location and permuted all the controls in between the controls. The way I structured the next part of the algorithm is, I start with a list of all the controls and reduce the controls in consideration step by step. This way if all the controls are traversed, we don't traverse anymore. This was the **second** optimization I implemented. I keep track of the best permutation, cost and the number of controls visited. For each permutation, I iteratively add time/cost for visiting the controls and check if the time exceeds the time limit allowed, if it does, I break out since it is going to be an invalid path. This was the **third** optimization implemented.

I go through the series of permutation to see if at least one path was found since the subsequent permutations have lesser number of controls as a part of the permutation (The itertools object contains the permutations that way). Finally, I return the best permutation, the number of controls visited and cost/time.

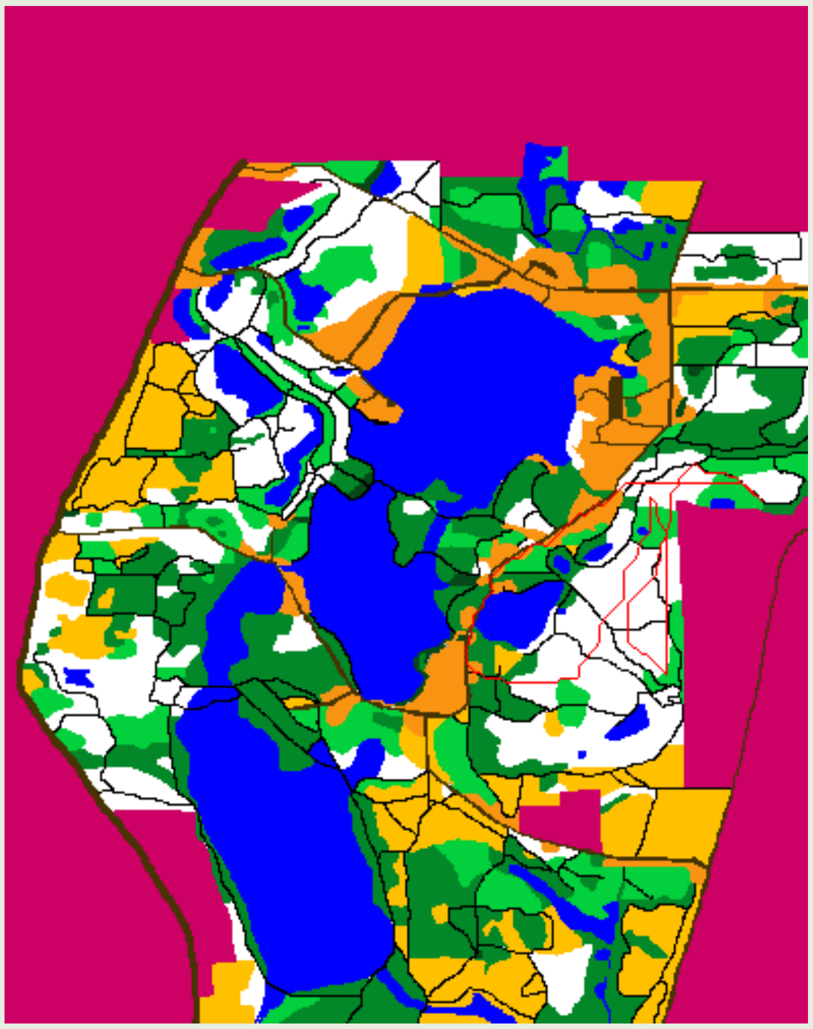
Apart from the *path_A_star_search(), cost_function(), successors()* and *is_goal()* functions that were used like the classic route, a few functions specific to the "ScoreO" event included:-

scoreo_route() - This is the calling function to find the optimal route and graph the route to traverse the score-o events.

create_adjacency_list() - This function forms the first basic step for the score-o problem. The function creates an adjacency list for each of the controls with the best cost to visit the other controls.

brute_force_permutations() - This function traverses through the series of permutations using the costs from the adjacency list by basically brute forcing to find an optimal path. This possibly would not be a scalable option however, for the given data we get a response in time.

ScoreO Event - 2009 s



PLANNING FOR SOMEONE ELSE

In order to plan for another person with different abilities, I modified 2 aspects, viz., the terrain costs that I was considering and heuristic function I was using. The heuristic function I assumed for planning the route was L_1 norm or **Manhattan** distance defined as the sum of the difference between the (x, y) coordinates.

A situation where my "planning for another person" differs from the route previously taken is clearly visible for the white route of the "Classic" event. On noticing closely we see that , previously, the route taken was to go beside the water body, however, in this situation due to the change in speed and heuristic function, the route taken goes around the water body.

Another difference we notice is in the amount of time taken to cover the routes is significantly greater.

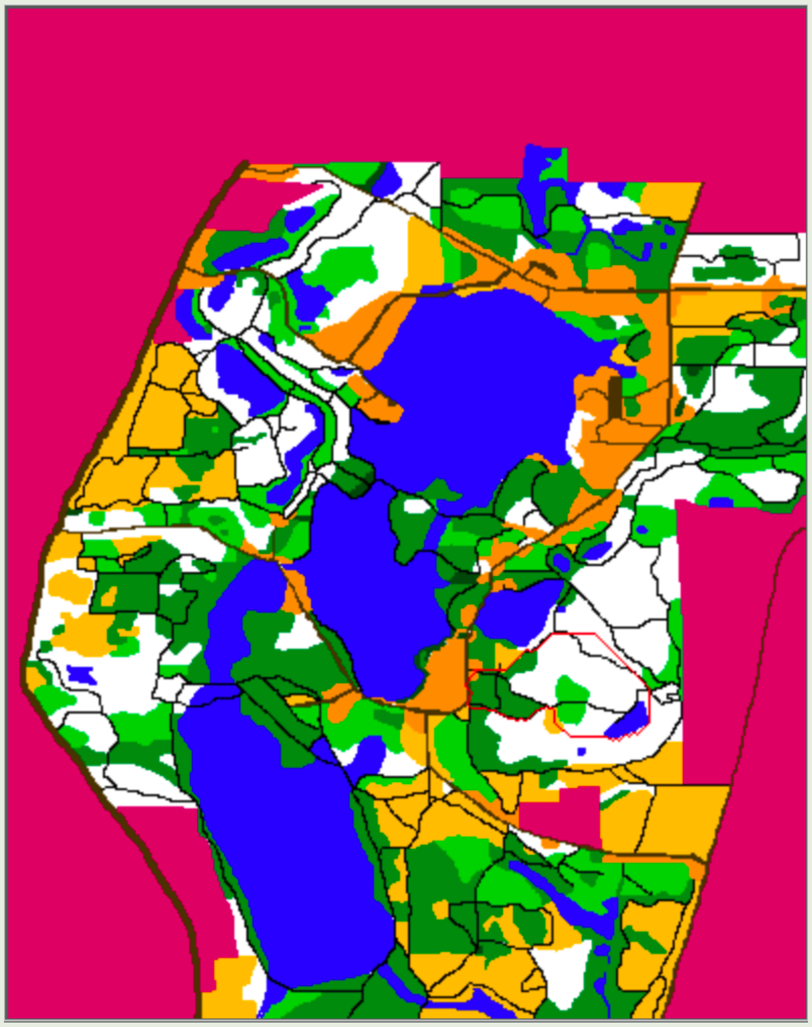
[I have commented out the code sections for "planning for someone else". In the route_finder_utils.py, the terrain costs for someone else is commented out and the *get_heuristic_distance()* is commented out. In order to run the program for "planning for someone else", swap the heuristic function and terrain speed values]

The terrain costs I modified assumed the following values:-

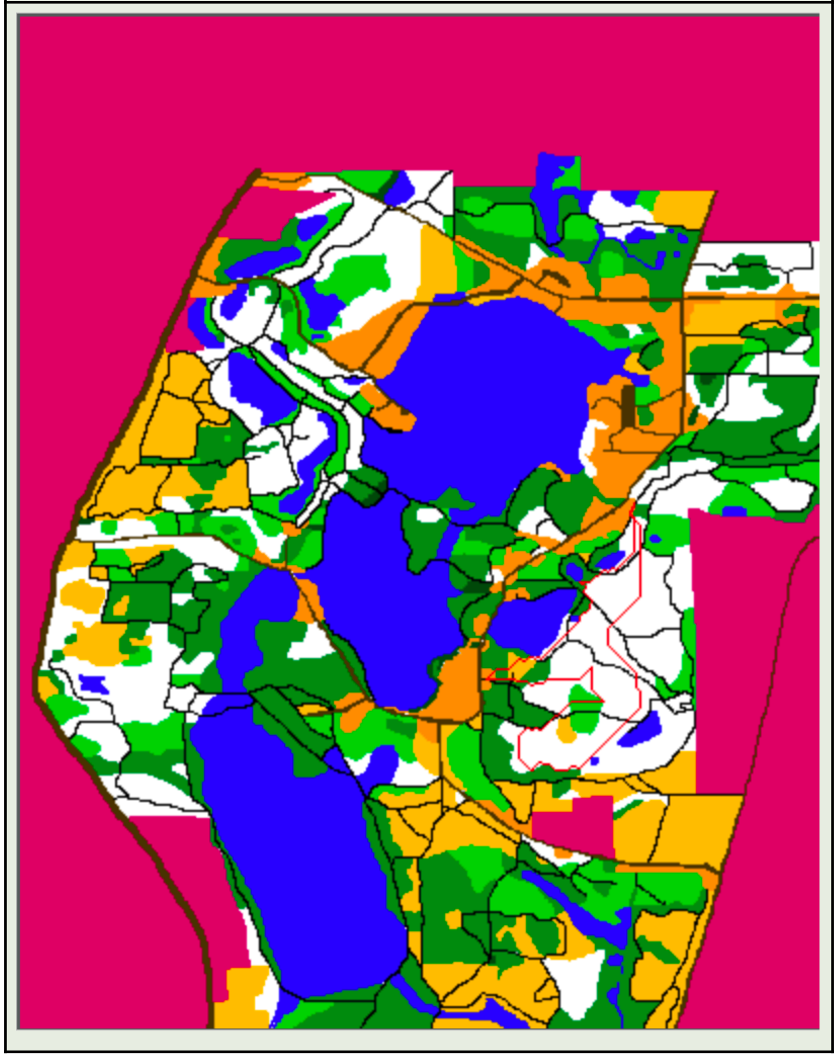
Terrain	**Speed (in m/s)**
Paved Road	3
Footpath	2
Open Land	2
Easy Movement Forest	1.8
Rough Meadow	1.4
Slow Run Forest	1.4
Walk Forest	0.5
Impassable Vegetation, Lake/Swamp/Marsh, Out Of Bounds	0

With modifying the values, the paths returned for the classic routes were the following:-

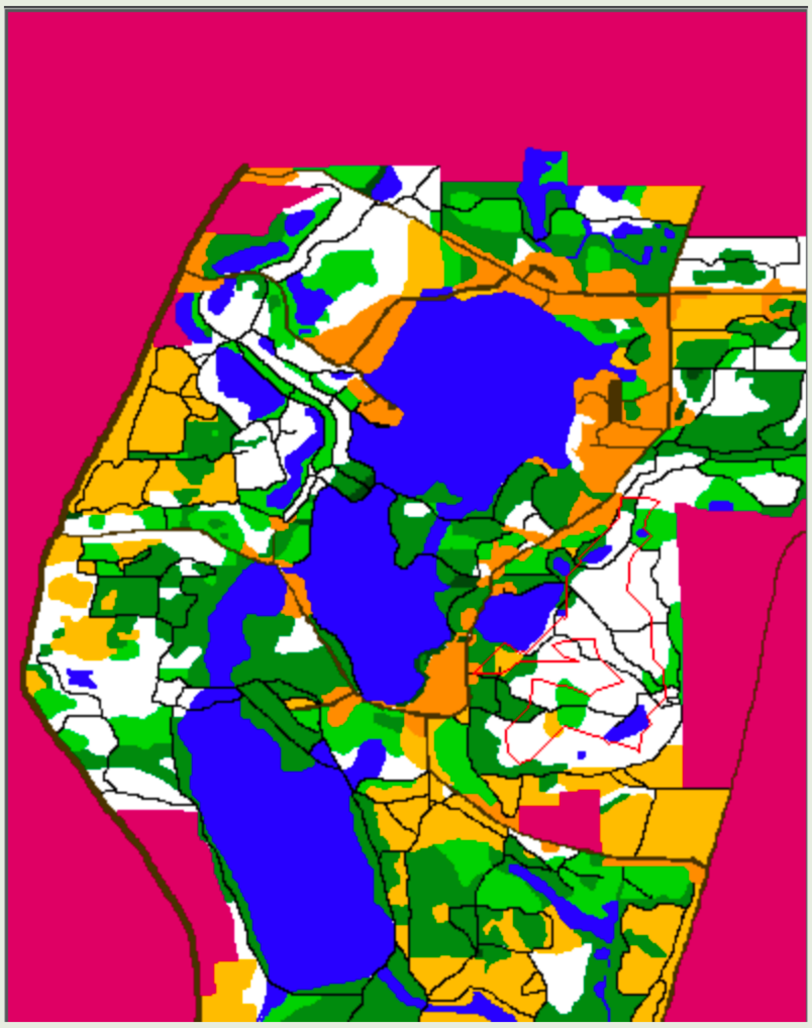
Classic Event - White Route (Planning for someone else) - 1537s

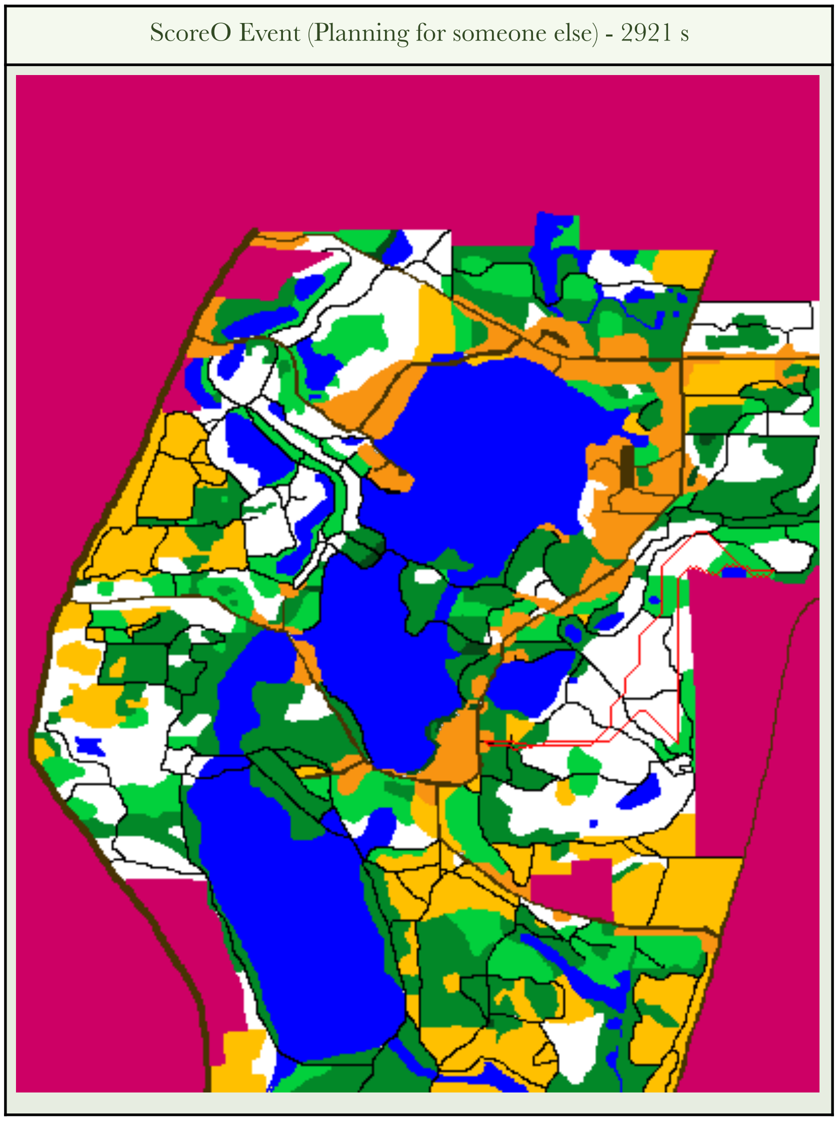


Classic Event - Brown Route (Planning for someone else) - 3015s



Classic Event - Red Route (Planning for someone else) - 3869s




ScoreO Event (Planning for someone else) - 2921 s

CONCLUSION

ROUTE	TIME TAKEN (in seconds)
White	763 s
Brown	1543 s
Red	2041 s
Score-O	2009 s

Summary of results

For the terrain speeds and heuristic function I picked, the time taken to traverse the route was as follows:-

ROUTE	TIME TAKEN (in seconds)
White	1537 s
Brown	3015 s
Red	3869 s
Score-O	2921 s

In “planning for someone else”, for the terrain speeds and heuristic function I chose, the time taken to traverse the route was as follows:-

In conclusion we can say that A * search is an optimal search algorithm because of using a cost function and a heuristic function. We can also further comment, looking at the above differences in the time, that the speed of moving through a terrain and the choice of heuristic function define how well does the search algorithm perform.

In a score-o event we are more likely to find an optimal path with minimum cost/time as compared to running a classic event for the same set of controls. However, score-o algorithm would not perform well if the number of controls increased as we are bound to have more permutations. In short, it would take more time to compute an optimal path in score-o, but the same would be take lesser time to traverse. On the other hand, while, traversing a classic event may take lesser time to compute, the time taken to traverse would be more.