**Overland Pathfinder**

Adam Gaudreau

COMP 5030 – Algorithms Final Project

Fall 2020

**Team**

I worked solo on this project.

**Project Description**

Getting from one place to another has never been easier today with all of the navigation applications available, such as Google Maps and Waze. These work great if your route is primarily on a road, however, most don’t perform great when the roads turn into dirt or vanish entirely. Overland Pathfinder was created to pick up where the others have left off, so to speak.

There are many useful scenarios for an off-road navigation application: search and rescue, hiking, exploring, wildfire rescue, and more. In most of these scenarios, you could use typical navigation tools such as maps and compasses, but it can be time consuming to plot a route that takes into account terrain, elevations, and other obstacles. When time is critical, you don’t want navigation to slow you down.

Overland Pathfinder is a proof-of-concept application that shows how off-road navigation can be approached as a web-based application. It’s as simple as clicking two points on the map to have a shortest-path route come into view.

**Technology Stack**

Since this is a web application, it’s built off the typical framework of HTML, CSS, and JavaScript. I used VueJS as a JavaScript framework to make development easier for the web interface. The core logic behind the routing algorithms is implemented in pure JS. I am also utilizing NodeJS to install packages for the website so that I can make requests to external APIs.

The only third-party code I’ve used other than the NodsJS packages is for the map, which is from Esri. In particular, I’m using the ArcGIS map for JavaScript, since implementing my own map module is beyond the scope of this project.

**Methodology**

There are three main steps to create a route after a user has specified the start and end points:

1. Create a bounding box,
2. Generate a flat map, and
3. Use Dijkstra’s algorithm (via Binary Heap) to find the shortest path.

*Create a Bounding Box*

A bounding box is a set of four coordinates that set the latitudinal and longitudinal boundaries of the flat map. When a user plots the two points, the minimum and maximum values are calculated, as well as the slope, to make sure there will be the appropriate number of vertexes, or nodes, in the graph.

Chart, radar chart

Description automatically generated

Figure 1: Bounding box displayed by red dots

*Generate Flat Map*

With the bounding box created, we can start building graph by adding a set of vertices and edges . Starting at the bottom left corner, we add a node upwards every 10 meters (by default), moving to the right 10 meters once we reach the top. The process continues until all vertices are created.

Background pattern

Description automatically generated

Figure 2: Flat map is created

Each vertex stores it’s coordinates on the map, it’s location in the flat map, and its elevation and terrain.

Each vertex has between three and eight immediate neighbors. To calculate the distance between them, we measure the ground distance between the two points, then make it a factor of its elevation delta. For example, if going uphill from one node to another, it will have a longer distance – vice versa if going downhill. This is one of the most important factors when plotting a route off-road; people will tend to want to avoid traversing tall hills, while prefer walking downhill.

The terrain is also fetched at each point, so that if the node is over non-traversable terrain (on foot) such as water, tar pit, volcano, etc., it will delete that vertex from the flat map. It will subsequently delete the edges to its neighbors so that the route can properly go around the obstacle.

With a set of vertices and edges calculated, we can finally calculate the route between the two points.

*Dijkstra’s Algorithm with Binary Heap*

I’ve implemented Dijkstra’s algorithm that utilizes a minimum priority queue, in this case a binary heap. The binary heap utilizes three key functions: insert (with priority), decrease\_priority, and extract\_min (which is essentially pop). Priority queues can lead to faster compute times than a basic queue, but that difference may not be noticeable once actually implemented.

Below is the pseudocode for Dijkstra’s with a binary heap.

Text, letter

Description automatically generated

Once *dist* and *prev* are found, we can backtrack through *prev* to get the shortest route from source to destination.

A picture containing text

Description automatically generated

Finally, here is the pseudocode for the binary heap I implemented:

A picture containing text

Description automatically generated

These are the core functions to get the shortest path. When the flat map is fed into Dijkstra’s algorithm, it will return an array of coordinates that the map API can use to lay the route as a layer over the current view. The route details will be displayed on the map, and the performance of the algorithm is displayed in the menu on the left side of the webpage.

**Theoretical Analysis**

Placeholder

**Dataset**

The map and subsequent API are provided by [Esri](https://www.esri.com/en-us/home). I got my elevation data from [USGS’s elevation service](https://www.usgs.gov/core-science-systems/national-geospatial-program/national-map). I used [GeoNames](https://www.geonames.org/) for the terrain data. All the data I collected was from trying out a many different routes and recording their statistics.

**Samples**

Please see the “sample\_runs” directory in the source code for higher resolution images of the program in action. All routes were initialed by clicking on the map at the start and end points.

Graphical user interface, map

Description automatically generated

Figure : Full route through foothills of California

Chart

Description automatically generated

Figure : Figure 3 but zoomed with flat map visible

Graphical user interface

Description automatically generated

Figure : Smaller route with flat map and bounding box

Text

Description automatically generated

Figure : Same as figure 5 with only bounding box

**Time Complexity Analysis**

As aforementioned, we’d expect this algorithm to run in super-linear time ()). To test this, I created 31 routes and recorded the runtime of the routing algorithm, the number of vertices, edges, total runtime, and distance of route. The data can be found in analysis.xlsx in the “docs” folder of the source code.

The following two charts are the results. As we can see in the Time Complexity graph, the runtime appears to be linear with respect to the number of vertices. Furthermore, since the runtime grows faster than the number of nodes does, it appears to run in slower time than a typical linear algorithm (. Thus, I conclude that this algorithm does indeed run in super-linear time.

The tests were conducted on my MacBook Pro (15-inch, 2017) with a 2.8GHz Intel Core i7 processor, 16GB of RAM, and a Radeon Pro 555 graphics card. The application was tested in the most recent version of Firefox.

To view the full images, see RouteVsRuntime.png and TimeComplexityAnalysis.png in the “docs” folder of the source code.

Chart, scatter chart

Description automatically generated

Figure : Count is X-axis, Runtime (ms) is Y-axis, blue is Vertices, orange is Edges

Chart, scatter chart

Description automatically generated

Figure : Distance (km) on X-axis, Runtime (ms) on X-axis

**Known Issues**

USGS is the only service I know that provides elevation data for free, however, it does not support bulk API requests. That means I have to make a single request for every node to get its elevation which adds a substantial amount of time to the route calculation. That being said, for all the performance measurements I referenced the time it took for the route-calculating algorithms, not the time it took to complete requests. This still allows us to evaluate the algorithm’s performance with this known issue.

Similarly, I need to make a single request for each node to get its terrain. GeoNames is a free service but it caps it at certain number of requests. For that reason the code is commented out, but it does work if you create an account and uncomment the code.

If you’re trying out the code yourself, please keep the routes on the short side (less than 0.5km). If I had a local DB filled with elevation and terrain data the app would perform significantly better.

3.     (10 pts) A theoretical analysis of the time complexity of your algorithms (5 bonus points for providing space complexity analysis).