Road Surface Area Estimation

FIT3036: Computer Science Project

Michael Franklin
Supervised by: Dr. Rasika Amarasiri

May 2018

Abstract

With the ever-increasing size of cities and their infrastructure, this project develops a tool that sources data from OpenStreetMaps in order to estimate the surface area of roads within a given bounding rectangle.

Keywords: Mapping, Roads, Surveying, OpenStreetMap

Word count: 4966 words

1 Introduction

Road construction and infrastructure maintenance is a large sector of the transport industry. As roads are generally owned by the Government, they are reliant on historical data and external surveyors to track the amount of road that they manage. When negotiating the cost and time requirements of road maintenance including the reconstruction of decayed roads, it's important to understand the scope and scale of such work; although there is no substitute for manual surveyance it might be useful for the owners to establish instant and cheap ballpark estimates. These estimates might also be a simple way for a council or road-owner to catalog the scale of their responsibility.

This project aims to fill that gap by sourcing data from web-mapping applications and providing configuration options such as individual road selection to allow the estimation of road surface area.

This road surface area estimation tool was proposed by Assoc. Prof. Andrew P Paplinski for the Computer Science Project (FIT3036), Semester 1 2018. This project was selected on the 13th of March, 2018 and was completed on the 28th of May, for a total project length of 11 weeks.

2 Background

2.1 Theory

There are a multitude of ways to get mapping data, such as running image analysis on satellite data, image analysis on rasterized map tiles, looking through historical records or manual surveyance. Large companies such as Google or Waze have difficulty getting accurate results just from satellite imagery, so use a combination of computer vision, google street view cars which has an element of manual surveyance, and local knowledge to ensure accurate maps [Madrigal, 2012]. OpenStreetMap is a collaborative web mapping application founded by Steve Coast which is similar to Google Maps on the surface, however its primary source of data is crowd sourced, inspired by Wikipedia's model [Lardinois, 2014]. To keep this surface estimation tool simple, it will rely on existing map data from OpenStreetMaps which provides free public APIs to determine the required outputs.

As the Earth is a 3D Spheroid (sphere-like), projections are used to map the surface into a 2D image. The geometry of the spheroid does not allow an exact or easy translation between two and three dimensions, so some properties such as accuracy in distance or perspective are traded for an aesthetically pleasant image. The most common projection method used by web mapping companies is the Mercator projection which attempts to compromise perspective, distance and area to create a final rectangular output. The Mercator projection became popular due to its easy understandability and capability to plot a straight-line course from one location to another, called a Rhumb Line [Britannica, 2018].

The World Geodetic System (WGS) uses the standard latitude, longitude coordinate pairs combined with raw altitude data and a nominal sea level to exactly specify a point on the Earth's surface. OpenStreetMap uses the the latest revision of the WGS (WGS84) and the Mercator projection for its data and subsequently generated tiled maps [Horman, 2016].

There are primarily two ways to calculate paths between two points, the Rhumb line and the great circle distance, the physically shortest distance. The rhumb line is a straight path between two points that cross each meridian (a line of longitude) at the same angle (constant azimuth) and appears to be the shortest distance on a map that uses a Mercator projection. The great circle distance is the most direct path on the surface of a 3D sphere. The geometric difference between these two can be seen in Figure 1.

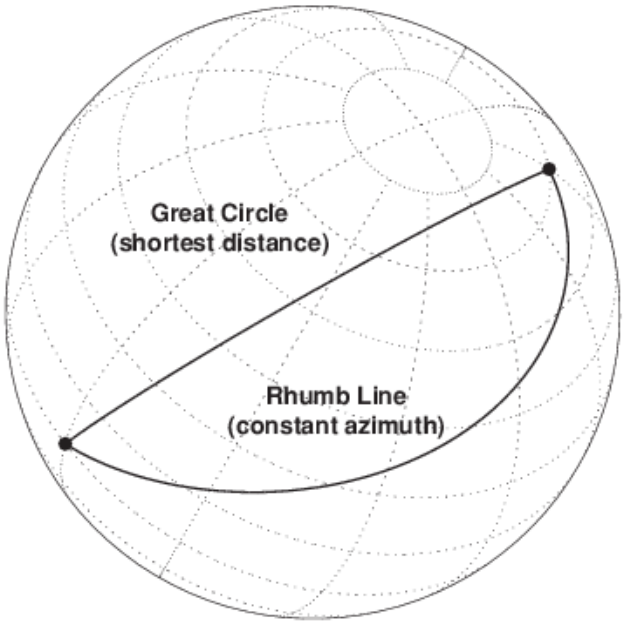

Figure 1: Difference between the Rhumb and Great Circle line on a sphere [Mathworks, 2018]

When determining distances between coordinates, the great circle (shortest) distance can be calculated by the modified Haversine Formula in equation (1), which uses trigonemtry to account for the changing distance between latitude and longitude lines. The Haversine Formula is independent of mapping projections.

$$d = 2r \arcsin \sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)} \tag{1}$$

where the following definitions and constants apply:

- d - distance between the two points in metres
- r - sphere radius[1] ($r_{\text{earth}} = 6,371,000$ m) [Williams, 2017]

[1] The Earth is a spheroid, so we take the volumetric radius and trade some accuracy for simplicity.

- φ_1, λ_1 - the Latitude and Longitude respectively of point 1
- φ_2, λ_2 - the Latitude and Longitude respectively of point 2

The final topic to introduce is the UI library React and Redux for state management. React is a JavaScript library in which views are composed of smaller React components that take inputs and returns a single JSX tag, similar to an HTML element.

React works on the basis that React Elements (components) are computationally cheap to instantiate while the HTML DOM is expensive to mutate. When the state of the application changes, React intelligently reloads the components in its virtual DOM, detects differences between this virtual DOM and the HTML DOM and writes any changes during a *reconciliation* process [Facebook, 2018].

Redux is an open source JavaScript library for state management. The general design pattern for React+Redux applications is to write narrowly focused components which can modify state by calling to Redux through actions and reducers, this informs React when state changes which can update it's virtual DOM and perform the reconciliation process if required. The flow of information through a redux application can be seen in Figure 2.

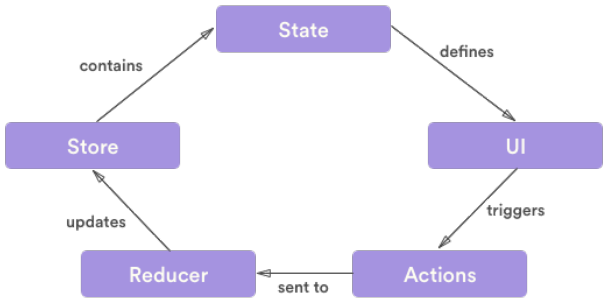

Figure 2: Redux state flow diagram [Krishna, 2018]

2.2 Risk and Risk management

There are three major risks that were identified early in the proposal process.

- (R1) - Project Deadline, this was addressed by ensuring progressive work was completed instead of rushing the project in the final few days. This was fortunate as there were two instances in which this risk was brought up.
- (R2) - Inaccurate calcluations were combated by finding examples of calculations and including them as unit tests.
- (R3) - The risk of data source providers such as OpenStreetMap or Overpass going offline was mitigated by finding multiple sources of this data to query. This risk was the motivator for

switching to using Overpass queries as it reduced the likelihood of reaching daily API request limits.

As introduced in (R1), there were two instances which impacted the timeline of this project. The first was planned surgery where my shoulder was reconstructed, this made typing difficult with one arm and when combined with rehabilitation, this made attending some classes difficult. Secondly, my car was caught in a thieve's crime spree, which resulted in a smashed window, buckled door and my bag stolen, which contained the primary laptop that I used to develop the application. Fortunately I was using GIT (and GitHub) to manage my source code so everything was backed up, however due to the inconvenience of sourcing a new computer, reparing the car and having to catch up on lost progress, I pushed my presentation back to the final week of the semester. It was fortunate that I was actively backing up my code, otherwise it would've put a lot of pressure to complete the project on time and maintain other workloads.

The loss of my primary computer provided an opportunity to ensure the application worked on multiple platforms (including Mac and PC) which was made much simpler by using Node as a development environment. Hence, the only resource requirement is an environment that has Node % NPM (Node Package Manager) installed, and an active internet connection to ensure the project can access the required data sources.

2.3 Timeline

The following timeline starts at Week 1 on February 26, and finishes Week 13 on May 27.

Table 1: Timeline of progress.

Week 3	Chooses the Road Surface Area Estimation project.
Week 4	Determines that OpenStreetMap is the likely source candidate for vector data.
Week 5	Creates the foundation for the server and a base UI.
Week 5	Surgery.
Week 7	Proposal document due, introducing concept of calculation modes.
Week 8	Impromptu presentation during Tutorial.
Week 9	Adds better inputs and address field to UI.
Week 11	Car break-in and laptop stolen.
Week 12	Project modified to work on PC, incremental fixes to the calculation, unit tests.
Week 13	Completes intersection algorithm, presentation and report writeup.
End W13	Submit.

3 Method

There are two sections to this project that are bridged using the NodeJS server, with all components implemented in JavaScript:

1. A user interface to select an area and display the results.
2. A processing algorithm that can fetch, process and return data to the user interface.

These two components are bundled into a single NodeJS application which can respond to web requests in order to, (a) return a user interface, (b) calculate the road surface area in a given bounding rectangle and calculation mode.

3.1 Models and Data Structure

The models this project uses are heavily influenced by three of OpenStreetMap's internal models, *tags*, *nodes* and *ways*. As OpenStretMap is a collaborative platform, all of these properties contain change set information which is discarded on load.

All elements can have tags, which provide context and more information to an existing object. These tags are dependent on the type of structure and the OpenStreetMap documentation should be consulted for more specific information.

A node is a single point that is defined by its latitude, longitude and ID. Nodes can represent parts of a road, or features such as speed bumps, water taps or other point objects.

A way is an ordered collection of between 2 and 2000 nodes to represent a larger ordered structure [OpenStreetMap, 2017]. When a *way* structure is provided by the server, it only lists the node identifiers in the ordered array, therefore some mapping is required to draw and manipulate a way. All roads are represented as ways and have a tag set to define their name, road type and other properties that we use to determine the width. All roads must have the *highway* tag present in their tags, even if it does not have value.

Figure 3 visually demonstrateds how the models are related.

3.2 Processing and Calculations

The major component of this project is the ability to estimate the road surface area within some confined coordinate bounds, this first requires acquiring the data from a web-mapping provider and then processing the result.

As discussed in the introduction, we query this data from a mirror of OpenStreetMap called Overpass. Overpass allows us to use a query to gain a narrow set of information, saving bandwidth and reducing the processing required by our server.

After the information has been returned by Overpass, we parse this to an internal JavaScript object. The schema for most of this information is not strict and can handle cases where Open-

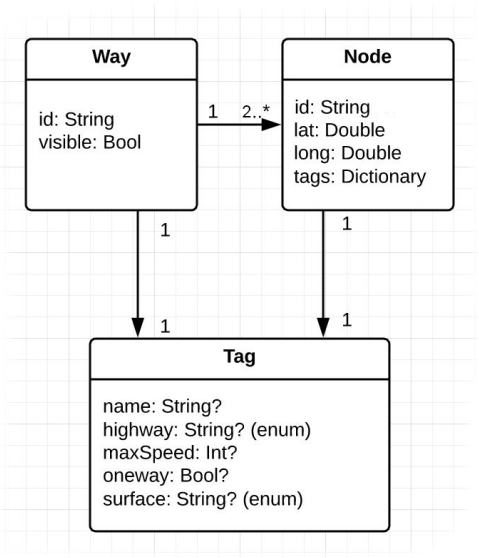

Figure 3: reduced OpenStreetMap models

StreetMap returns less or more than what is requested.

We calculate the surface area of each road independently of each other in a loop, and then return all this processed data to the server. This loop is highly parallelizable due to the preprocessing of information. This calculation sequence of events can be seen in the following sequence diagram.

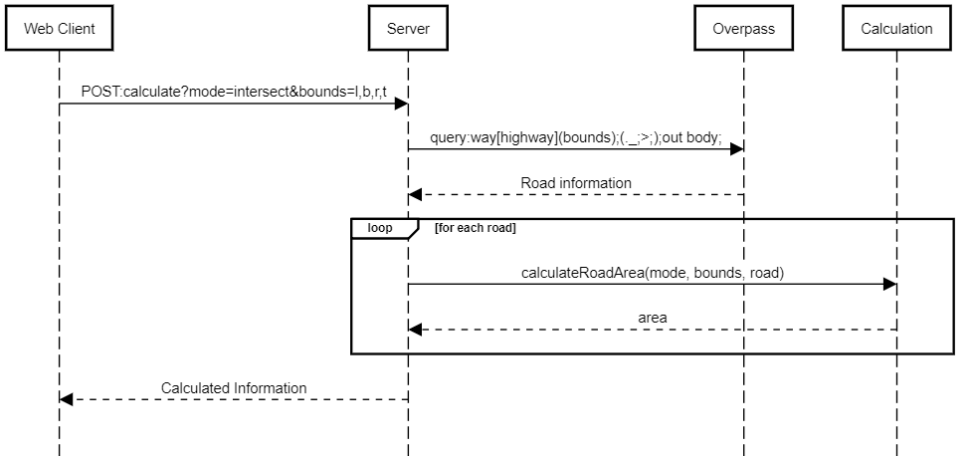

Figure 4: Sequence diagram of the calculation request

The following query is sent to the Overpass API to get all the roads within a bounding box:

```
way[highway](${bottom},${left},${top},${right});(._;>;);out body;
```

This gets all of the *ways* inside the bounding box with the required *highway* tag, and all information required to understand this information [(._;>;);], which is primarily the nodes [Overpass, 2017].

Overpass returns additional information outside the bounding box to give us enough context to correctly draw the map, so we introduce 3 calculation modes to allow the user to decide how to handle this:

- Include: Include all information returned by Overpass in the calculation, do not perform any filtering.
- Intersect: Intersect the road at the bounds by calcluating an intemediary point at the border and disregarding all other points outside.
- Truncate: Truncate the road to the nearest intersection.

We look at these three modes separately.

3.2.1 Include calculation

No filtering here is required, so we return all the ordered points.

3.2.2 Intersection Calculations

This section uses a combination of two algorithms:

- Calculating the intersecting point.
- Reducing the list to only points inside the bounding box, and calculating the intersecting point of the last point inside, and the first outside.

The first algorithm takes two points and a bounding box and returns a third point, where we require that one point be inside the bounding box and one is outside. We start by plotting an example of the problem in Figure 5.

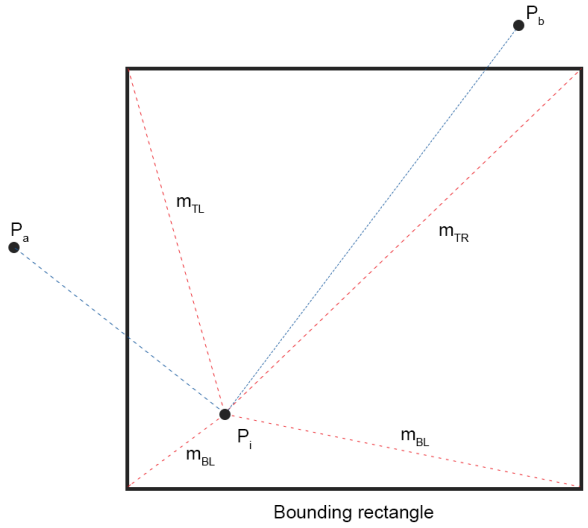

Figure 5: Diagram of how intersections between two points and a bounding box are performed by comparing gradients where the solid black lines are the bounding box, P_i is a point inside the bounds, P_a and P_b are two points outside the box

The major problem to solve is which edge of the box the connection between two points fall, by knowing which edge is intersected, we have one known latitude or longitude point in which we can easily calculate the other. We treat the pairs P_i, P_a and P_i, P_b separately, and draw dashed lines between the pairs of points. Remembering that these coordinates the latitude / longitude pairs, the dashed line is the rhumb line between two pairs. We additionally draw red dashed lines between the inside point and the corners of the gradient.

Let the gradient between P_a and P_i be m_a and the gradient between P_i and P_b to be m_b as in Figure 5. We first consider if the point outside the bounds is to the left or the right of the point inside the bounds:

- If it's to the left (P_a):
  - If $m_a \leq m_{TL} \implies$ Top intersection.
  - If $m_a \geq m_{BL} \implies$ Bottom intersection.
  - Else $\implies$ Left intersection.
- If it's to the right (P_b):
  - If $m_b \geq m_{TR} \implies$ Top intersection.
  - If $m_b \leq m_{BR} \implies$: Bottom intersection.
  - Else $\implies$ Right intersection.

Now that we know which edge of the bounds the rhumb line travels through, we also know one of the coordinate values, that is either the top or bottom latitude of the bounds (if it's a top or bottom intersection), or the left or right longitude of the bounds (if it's a left or right intersection). We can then use the gradient m to calculate the other coordinate by:

$$\text{newLat} = P_i.\text{lat} + (\text{left} - P_i.\text{lon}) \cdot m,$$
$$\text{newLon} = P_i.\text{lon} + (top - P_i.\text{lat})/m,$$

where top / bottom are interchangeable, left / right are interchangeable and P_i.lat / lon represents the latitude or longitude of the P_i point.

Secondly to filter out any points outside the bounds, we start at one end of a list of ordered nodes and iterate along until we find the first one inside the bounds. Next we calculate the intersection via the previous algorithm of the first point inside the bounds and the previous node, remove all those points outside the bounds, and insert the new intersection at that place. (If the first node is inside the bounds, then skip this step).

Similarly, we start at the end of the list, and iterate back until we find one in in the bounds, we then calculate the intersection of this node and the one we searched before, remove all the elements outside the bounds and add the intersection in its place. (If the last node is inside the bounds, then skip this step).

This algorithm has the consequence that if both the first and the last node are inside the bounds, then no filtering will occur.

3.2.3 Truncation calculation

This has very similar goals to the intersect algorithm, however instead of calculating the intersection point, we attempt to find a node where another road is connected to. We start by building up a relationship between a node and the list of roads it's connected to when we are processing the response from Overpass.

Then when we are required to filter the ordered nodes and the edge nodes are outside the bounds, we can simply check through the ordered nodes until we find one that is both inside the bounds and has more than one road attached to the same node.

Like the intersect algorithm, this also has the consequence that if both the first and the last node are inside the bounds, no filtering will occur.

3.3 User Interface

The user interface is a single page application built using React and Redux. The front-end application should be able to make web-requests to the server and process the response by summing up a list of numbers given conditions set out in the functionality.

It should implement the functionality listed out in the proposal document, this includes:

- Have a draggable map which can update the bounds

- Type in coordinate bounds to update the calculation
- Search for an address
- Configure the calculation by exluding some roads or types of roads

Per the React methodologies, components should be kept small and have narrow purposes to avoid nesting complexity together. This section is left open for interpretation as there are a number ways a user interface could be designed, styled or presented.

4 Results

The server framework implementation was straight forward given the simplicity of its purpose. The project uses a number of modules to faciliate and improve the development environment. We discuss the environment modules here, and more specific modules in their sections below.

- `express` allows a NodeJS application to accept web requests.
- `babel-polyfill` compiles any higher-level JavaScript into browser readable code, it also allows the use of the ES6 JavaScript specification for development.
- `axios` simplifies syntax for external web requests, similar to jQuery's `$.ajax`.
- `cross-env` allows specific environment variables and improves support for cross-platform server hosting.
- `expect` and `mocha` are two libraries that make the unit testing process simpler.
- `eslint` enforces better JavaScript patterns and conventions.

4.1 Additional Models

In addition to those models specified in the methodology, a `ReducedNode(id, lat, lon)` class was created to represent an OSM node, this class provides additional methods to check whether itself exists within some bounds and has specific unit tests for this functionality.

4.2 Fetching data

Fetching the data from Overpass was straight forward after the query was constructed and tested. The only export format for OpenStreetMap is XML, this output is the same for Overpass, however it was later discovered (too late to implement) that there is a way to get a JSON export from Overpass. This XML export required parsing using the `xml2js` library with some additional scripts to clean up artifacts from the conversion such as one-to-one relationships mapped into arrays or the way properties are mapped onto an object, for example:

```
<node id="34016489" lat="-37.8550290" lon="145.4185842"/>
```

gets mapped to:

```
{                                    {
"$":  {                               "id":  "34016543",
 "id":  "34016543",       then        "lat":  "-37.8511023",
 "lat":  "-37.8511023",     to        "lon":  "145.4235115"
 "lon":  "145.4235115"                }
}
}
```

Across 10 runs of an intermediately sized Monash Clayton area, the query to Overpass took an average of 1300ms to return the XML response of 39KB. The server parsed the response in 150ms, and calculates the area of 450 roads in 8ms. The response from the server is JSON of 25KB to reduce any additional processing by the client. If bandwidth was an issue, the response could be made more efficient by distributing some processing requirements from the server to the client.

4.3 Calculation Implementation

The implementation of the Haversine Formula, the width detection and the general area calcluation was successful, which unit tests validate. As the ways reference the nodes by their ids, a mapping was performed using dictionaries which have an `O(1)` time complexity for lookups.

The implementation for calculation modes was simpler than anticipated, rather than creating three similar implementations of the calculation, only one small filter step in the larger calculation chain was required. This made the process easy testable and avoided code reuse.

All of the algorithms have a space complexity of O(N+R) where N is the number of nodes, and R is the number of roads, with a worst case time complexity of O(N + R), as there is very little overlap between roads and nodes (except ordinarily the edges of the ordered list).

4.3.1 Intersect filtering

The intersect filtering was successfully implemented, as can be seen by the the following generated graph which by inspection determines that the intersection point was correctly calculated.

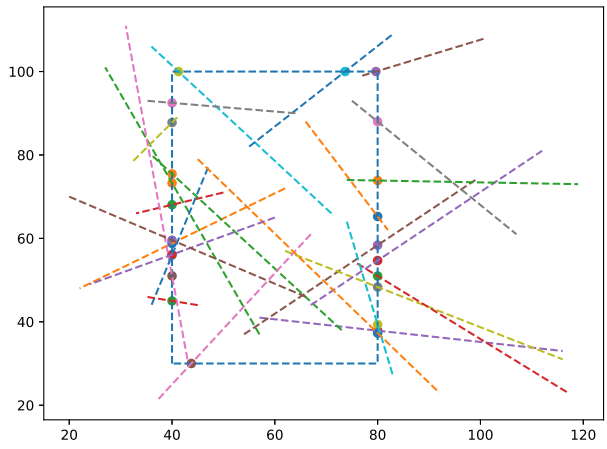

Figure 6: Generated plot to test the intersect algorithm

4.3.2 Truncate filtering

The truncate filtering had a similar implementation to the Intersect filtering and the algorithm worked as expected by truncating to the nearest point at which two *way*s are joined, however in reality the data that the API returns makes determining where two distinct roads join very difficult. For example, some roads are split into multiple ways as they provide different tags, if the road widens at a specific section or the speed limit changes, this algorithm thinks that there is a distinctive road change.

4.3.3 Comparing calculation modes

We compare the Monash Clayton campus for the bounds (145.12488,-37.91435,145.14162,-37.90758), and get the following three values with all types and roads selected:

- Include: $254,544.4$ m^2
- Intersect: $208,255.1$ m^2
- Truncate: $206,794.4$ m^2

We plot the three outputs of the filtered nodes to visually compare how effectively they worked for a fairly complicated data source.

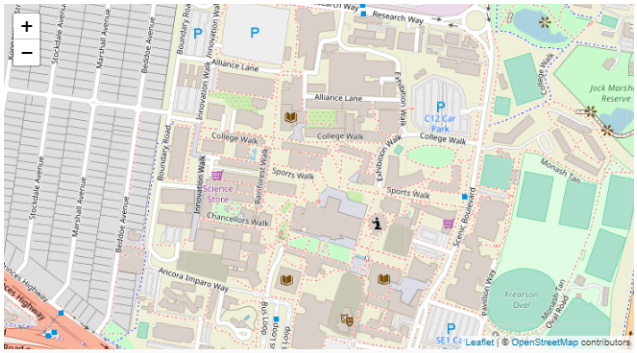

Figure 7: The tiled map that is the bounds for the following plots.

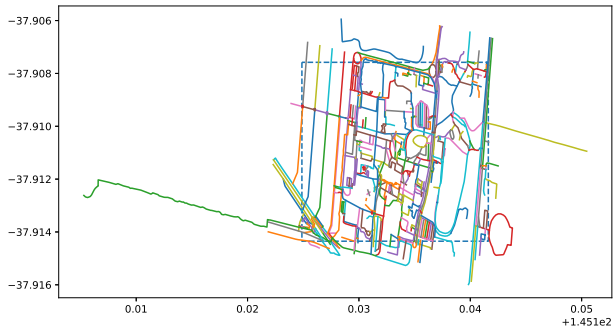

Figure 8: *Include*: A plot of the roads returned by OSM with no filtering performed, the bounds are a dashed rectangle in the center of the image.

We clearly see that the three algorithms all give separate meaning to the calculation, and obey the rules that they were given. The visual interpretation of the three graphs show that it is difficult to determine structures such as loops that start and end inside the bounds such as the round-about

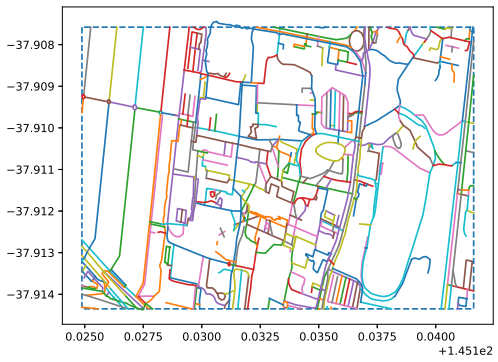

Figure 9: *Intersect*: A plot of the roads returned by OSM filtered to the bounding box, with an intermediary point calculated at the dashed bounds.

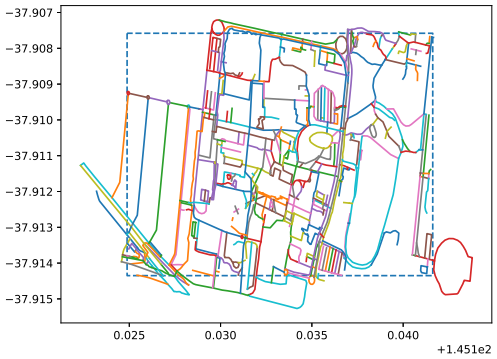

Figure 10: *Truncate*: A plot of the roads returned by OSM, truncated to the nearest intersection in the dashed bounds.

hour

in the top. It's also difficult for us to quantiatively determine the accuracy of the measurements, but in an intuitive sense, *include* sensibly returns the highest result, and *intersect* & *truncate* have similar results for this data set.

There are no specific real-world test cases with quantitative results as this does not give an effective test of the application but the quality of the data from OpenStreetMap. There are unit tests which partially cover the calculations described in the methodology, more information about these unit tests can be found in the Test Report [Franklin, 2018a].

4.4 User interface

The user interface had a lot of functionality to fit in, but I believe it has been designed in such a way that the functionality is intuitive. Due to the complexity in setting up the design patterns and transpiling through babel, the React project was initially forked from Cory House's (2017) pluralsight-redux-starter, which is the starting template used in the PluralSight course of the same name [House, 2017]. There is little to no code still remaining but fragments of the configuration.

The result of the calculation is prominently displayed in the top left-hand corner of the screen, with the immediate configuration options below it. By default, a number of *types* are deselected because I have pre-identified them to not likely be roads, but pathways or cycleways. The screenshot in Figure 12 shows all of the options ticked to align with the comparison between the calculation modes.

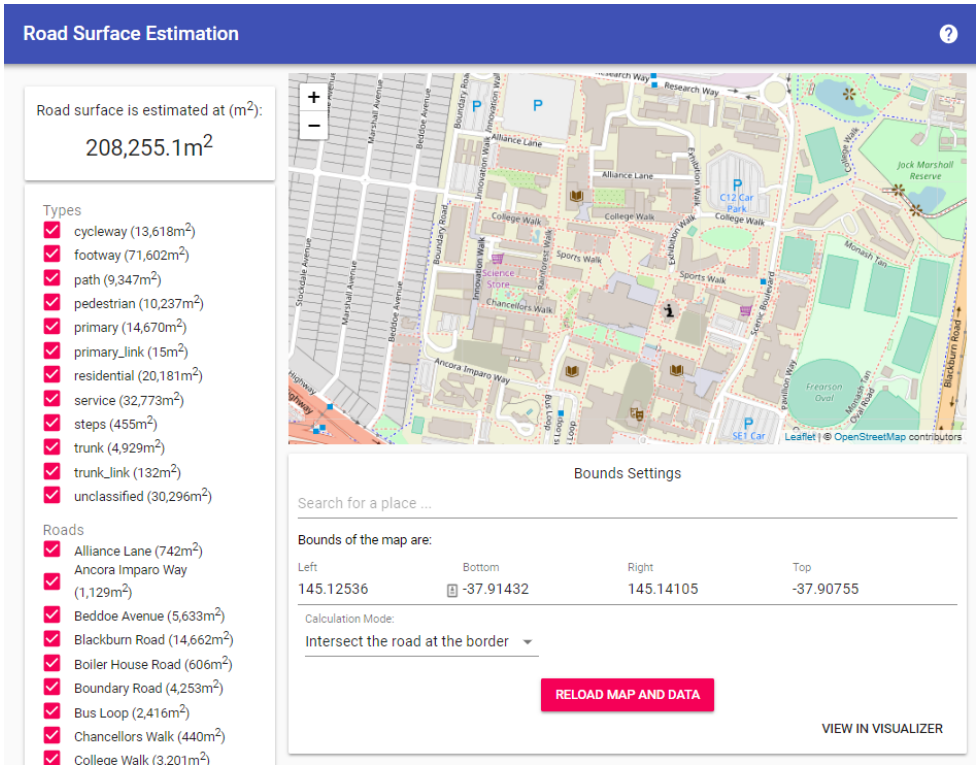

Figure 11: Screenshot of the user interface available through the web-browser

The interface used a number of Material-UI elements for a consistent and recognisable styling, and used React-Leaflet for the interactive map (which uses OpenStreetMap for the image tiles). The place search uses Google's Place Autocomplete API [Google, 2018], this is working as of 28th of May, 2018, howver Google Maps has some planned API changes in response to their Google I/O conference this year.

All required functionality listed in the proposal document and methodology section are implemented, and additionally an extra button with the text *VIEW IN VISUALIZER* allows the user to view the data returned by Overpass by visiting their Overpass-Turbo with the query prefilled.

An *About* page is presented when the question mark icon in the top right-hand corner of the screen is tapped which lists some of the modules used to create the page.

5 Analysis & Discussion

5.1 Calculation

The implementation of the analysis section was successful in its initial criteria, both the intersect and truncate mode correctly filtered nodes out of ways that fit the criteria to be removed. We mentioned three results, one for each mode for the calculation of the Monash Clayton campus and surrounding infrastructure. As expected, we saw that *include* had the highest measurement, with both the *truncate* and *intersect* methods within 1% of each other, this is likely coincidence in the area and the bounding box that was selected. We expect to see more roads completely removed by the truncate method in suburban areas where there are lots of short roads with many intersections, where the intersect method will only cut off these roads to the border.

In more examples, the truncate method does in fact reduce the number of roads and calculates a noticeably smaller result that the intersect method.

5.2 User Interface

The basic user interface was decided on fairly early, but had many incremental changes and tweaks to ensure that functionality was grouped together logically and its functionality is intuitive. The UI works best on screens with a resolution greater than 1024x768 so the configuration options display nicely alongside the map. The way React handles state made handling and displaying errors easier to manager. If an error occurs during the calculation, the application try its best to find the error and display it to the user with a *RETRY* button.

An issue with the user interface is the bounds is dependent on the aspect ratio of the map, which depending on the browser window and screen size can prohibit making exact replications of calculations through the UI.

6 Future Work

6.1 Calculation

Future development would likely see a higher number, and better depth to the unit tests to cover more of the application. As it currently stands, a majority of calculation components have some unit tests, but more diverse tests might better catch edge cases and reduce the likelihood that future changes break existing functionality.

A quick efficiency boost to the calculation could be parallelizing the area calculation boost, though the benchmarks through the results suggest that this isn't a priority bottle-neck.

6.2 Server improvements

The server had a fairly quick implementation cycle, this resulted in some inefficiencies in the way data was transported to the client, this could be cleaned up which might speed up the user interface on larger data sets.

For future development, it would be useful if the server could export a report given the bounds, and not just calculation data for the client to interpret, this might improve reusability and usefulness of the application

6.3 User Interface

The user interface is a good foundation that satisfies the specified criteria, however there are a few features that would be useful or could be improved.

- When selecting a large amount of roads ($>$ 400 roads), the application can become slow to recalculate the estimated road surface area, this was replicated on an area of (4.5 x 3) km, which is significantly greater than the specified area required.
- Given the different calculation modes, it would be useful to see an overlay of the data on the map that could be toggled on or off.
- Provide more visual aids such as interactive plots to explore the data.
- Expose more raw information obtained from OpenStreetMap such as road surface type (sealed, gravel, dirt, etc), and provide a method for users of the application to correct information in OpenStreetMaps to improve the accuracy of results.

7 Conclusion

This project took the road surface area brief and explored the different avenues that could be followed to estimate the road surface area. OpenStreetMap was chosen to be the primary source of data due to its free and public API, and flexibilty to allow users to update the accuracy of their own maps, increasing the accuracy of the estimates and improving the quality of data for all users of OSM. A software system was built to automate this estimation process with built in tools to allow the user to specify the calculation and inclusion modes for roads.

8 Bibliography

References

[Britannica, 2018] Britannica, E. (2018). Mercator projection.

[Facebook, 2018] Facebook (2018). Virtul dom and internals.

[Franklin, 2018a] Franklin, M. (2018a). Computer science project - road surface area estimation.

[Franklin, 2018b] Franklin, M. (2018b). Test report.

[Google, 2018] Google (2018). Places api for web.

[Horman, 2016] Horman, J. T. . C. (2016). Projections/spatial reference systems.

[House, 2016] House, C. (2016). Building applications with react and redux in es6.

[House, 2017] House, C. (2017). pluralsight-redux-starter.

[Krishna, 2018] Krishna, S. (2018). Is redux a javascript framework or a javascript library for react?

[Lardinois, 2014] Lardinois, F. (2014). For the love of open mapping data.

[Madrigal, 2012] Madrigal, A. C. (2012). How google builds its maps — and what it means for the future of everything.

[Mathworks, 2018] Mathworks (2018). Rhumb lines.

[OpenStreetMap, 2017] OpenStreetMap (2017). Elements.

[Overpass, 2017] Overpass (2017). Overpass api / language guide.

[Williams, 2017] Williams, D. (2017). Earth fact sheet.

9 Appendices

9.1 Requirements

The application has the following requirements:

- An internet connection
- NodeJS and Node Package Manager
- Windows or MacOS

9.2 Running the project

The NPM module `node-gyp` must be installed globally, via:

```
npm install -g node-gyp
```

In the project folder, you only have to run:

```
npm install
```

once to install any node dependencies, and then:

```
npm start -s
```

to run the project. You should get the following messages:

- `Started on port: 3000` - The server is listening out for requests.
- `17 passing (..ms)` - The unit tests successfully completed.
- `Clean (00:n:nn)` - There are no ESLint warning about the JavaScript.
- `webpack built {guid} in ....ms` - Babel successfully transpiled the React application (user interface), which can be visited through `http://localhost:3000`.

As mentioned in the start up phase, there are a number of unit tests that are automatically performed on start of the application, these are further discussed in the test report [Franklin, 2018a].

9.3 Testing the interface

All user facing functions can be used through the web user interface, or the functions can be tested manually through two POST requests, both of which require a bounding box (`bbox=left,bottom,right,top`) and a mode (`mode=intersect`):

- `/api/calculate` - The main processing function that returns a list of *roads* with their area and types (may have mutiple entries for a single road), a *roadNames* object with the road name, it's area and the types it corresponds to, and a *types* object that has the road type with the area of all the roads of that type as its value.

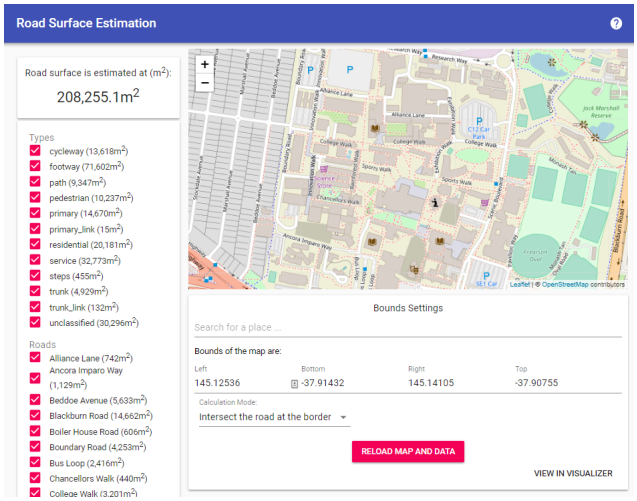

Figure 12: Screenshot of the user interface available through the web-browser

- `/api/coordinates` - A list of ordered coordinate pairs, one list for each section of road returned by the Overpass API which can be plotted. This was used to test how each mode reacted to the bounds, mainly for verification purposes and was out of scope for the primary project.

The following is a screenshot of the interface.
On first load, the Monash area should be automatically loaded, and the calculation should automatically begin to load.

You can select a new area by:

- Scrolling to a new area and then clicking the *RELOAD MAP AND DATA* button.
- Typing a place into the *Search for a place ...* field, and selecting one of the suggested places, see Figure 13.
- Typing in new bounds and clicking the *RELOAD MAP AND DATA* button, see Figure 14.

The calculation mode can be changed by tapping on the field *Calculation mode* and selecting a new result, by default it will have selected the *intersect* mode with the displayed text, 'Intersect the road at the border', see Figure 15.
Roads and Road Types can be toggled via the menu on the left-hand side of the screen. Tapping any checkbox will automatically calculate the new sum, see Figure 16. Some types are automatically deselected as they are not roads.

If an error occurs, the application will attempt to find the source of the error to display to the user, it will also display a *RETRY* button as seen in Figure 17.

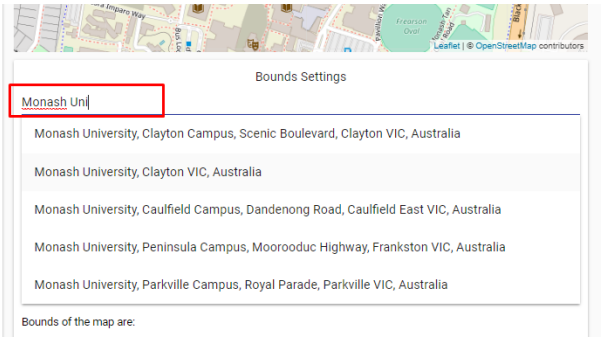

Figure 13: Screenshot of typing *Monash Uni* into the place search bar with 5 suggested results. Clicking on a suggestion will automatically reload the map and calculate the new bounds

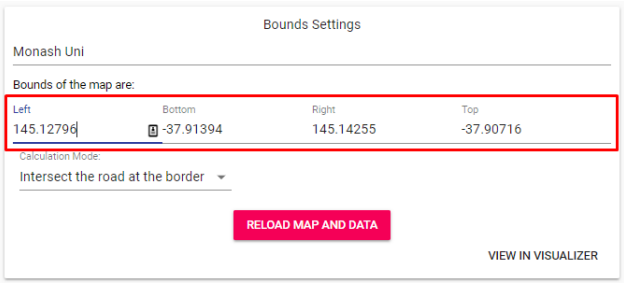

Figure 14: New bounds can be selected by typing in four new bounds (left, bottom, right, top) and tapping the *RELOAD MAP AND DATA* button.

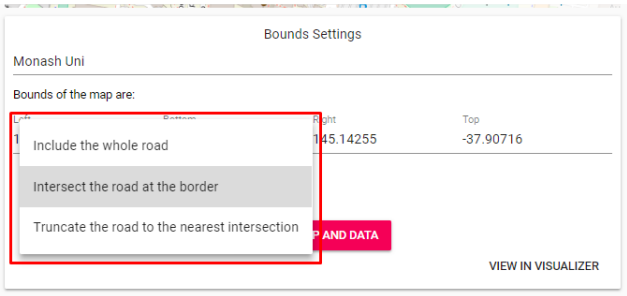

Figure 15: The red highlighted box shows the three calculation modes that can be selected in this project.

9.4 Testing tools

A majority of the testing was performed by unit tests from within the application, however two Python scripts small scripts were developed to:

1. `fit3036-intersection-test.py`: Test the intersection algorithm across a variety of gener-

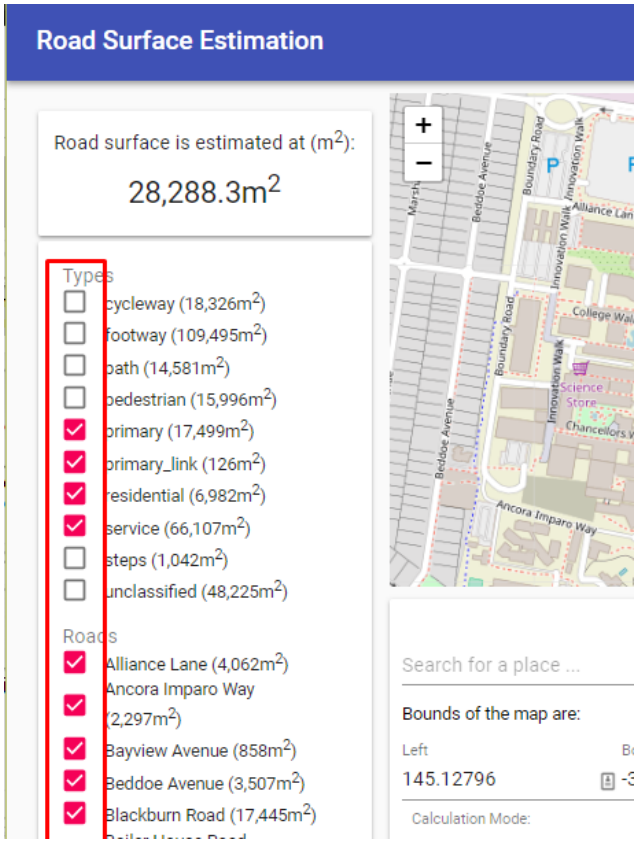

Figure 16: Changing which roads and which road types are included in the final result is easy by clicking the checkmark next to the name of the item you want to toggle.

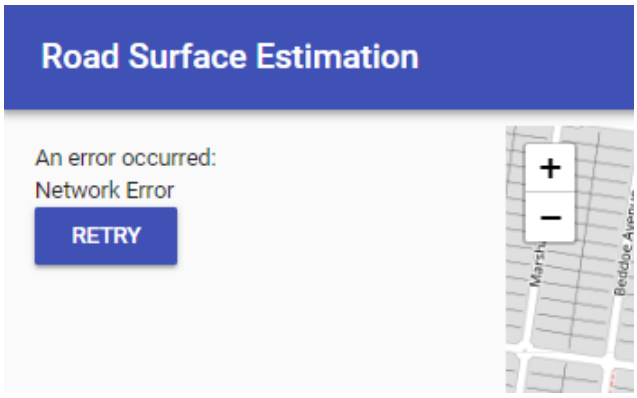

Figure 17: A *RETRY* button occurs if an error occurs.

ated examples (Figure 6).

2. `fit3036-mode-plot.py`: Plot the ordered coordinates returned by the software to visualise (qualitatively) how effectively a mode's filtering worked (Figures 8,9,10).

Both of these scripts require MatPlotLib with the first additionally requiring random. It's recommended to use the Anaconda distribution of Python to ensure these modules are installed correctly.