Executive Summary

Geographic Information Systems provide geographic referencing and location analysis. Businesses use them for spatial and predictive analysis and also for cost-saving purposes. Municipalities use GIS to plan more efficient routes for public transports, since buses usually involve stopping at multiple locations in a city. A commercial application involves courier companies who want to get their deliveries completed using an efficient delivery route to reduce travel expenses. A personal application involves tourists looking for city hotspots and points of interest.

Objectives

Our main goal is to make our GIS be targeted towards a wider number of users, ranging from personal to commercial users. This is emphasized upon by building an interface with a fast response time and one that is self-explanatory for first time users. Users range from tourists to courier companies and the municipality. This goal also has a larger impact on society. Street directed GIS reduce traffic congestion and fuel expenses and save time as the paths provided are very optimal.

Intentions and Benefits

Our interface is currently a street directed GIS with path-finding features. Features include being able to find a path through either two points or multiple points. This can be done by either clicking points within a map or entering destination names in search bar that is equipped with both auto-correction and auto-completion. Routes are shown both visually and with textual directions. Our goal is reinforced through the variety of features that we implemented, taking into account the wide range of user preferences.

Future Development

We decide to enhance our design into a multi purpose GIS software directed towards more users, which provides map layers for humanitarian organizations such as Red Cross. Using these maps they can have a better understanding of the disastrous areas while preparing logistics for an outbreak. The second feature provides power plant companies with a regional and global map of renewable energy resources, so that they can be used to find the best region for constructing new power plants, such as wind turbines and solar power stations. Providing a map with multiple layers can widen the range of our users.

1. Introduction

This document is a report to present our team's final GIS design and to describe areas of GIS applications that have potential for development, such as humanitarian organization and renewable energy. It illustrates the viability of our design in the marketplace, in which our primary users will be tourists and courier companies. It also outlines a high-level view of our code architecture such as the importance of the files m3.cpp and m4.cpp compared to the other files. It describes many algorithms such as those used in the courier route optimizer which involves a mix of Dijkstra and k-opt.

This document also explains specific design features that reinforce our core goal, which is to target a wider range of users by meeting their personal preferences. The implemented features which allow us to reach our goal are:

- The path finding tool can be used both commercially (by courier companies) and personally (by tourists) by allowing users to find a path through either 2 points or multiple points.
- Due to differences in user preferences, both visual representations and textual navigations of the path are shown.
- To find a path between points, users can either click on points in the map or type the destination in a search bar which is equipped with an autocorrect and autocomplete feature.

2. Technical Overview

Our code is separated into files to reduce the complexity of the project. This allows for future enhancements to the project by making modifications to various classes such as adding data members.

A. High-level View of Code Architecture And the Files

The source of the map data is OpenStreetMap, an open geographic database. OSM data were parsed and loaded into libstreetdatabase.

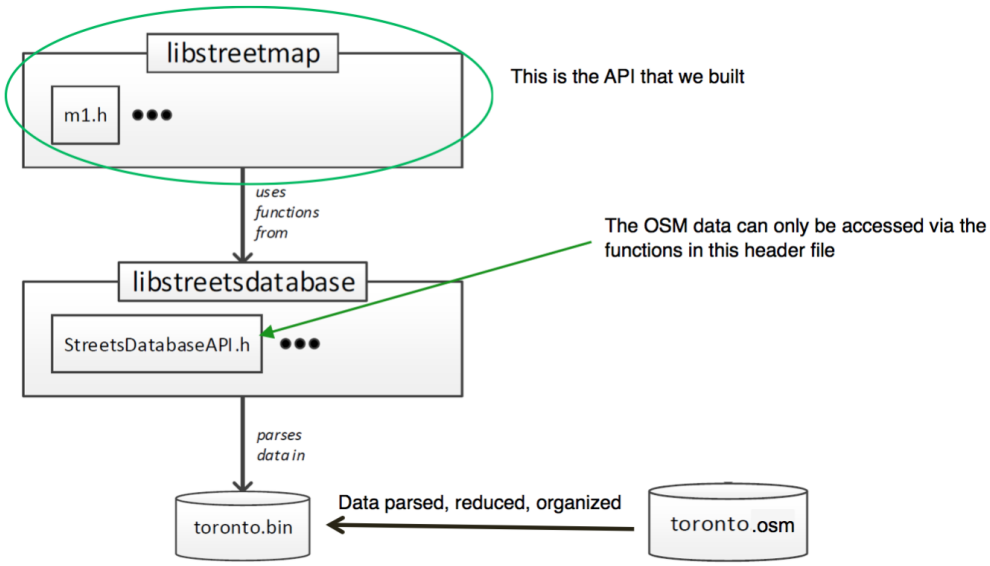

Figure 1: The above figure shows the API layers in this project. Libstreetmap is a folder which contains all of our files. The OSM data can only be accessed via the functions in StreetsDatabaseAPI.h. This is the lower API given to us. [1]

- **m1.cpp**

  Using the provided API, OSM data was loaded into our own data structures by using functions in StreetsDatabaseAPI. This was done to avoid constantly retrieving data from the lower API as that would slow down the program. Our data structures consist of hash tables to maintain a read and write complexity of O(1). In addition to loading the data, helper functions were written to access data from these data structures. These functions include finding a street length and finding the street names of an intersection. The loading process involves going through all the intersection IDs, the street segments, and the street IDs once and storing their respective information like the street segment IDs and coordinates of curve points.

Two of the data structures used are shown in the following figures. The other data structures follow an identical format but with different keys and values.

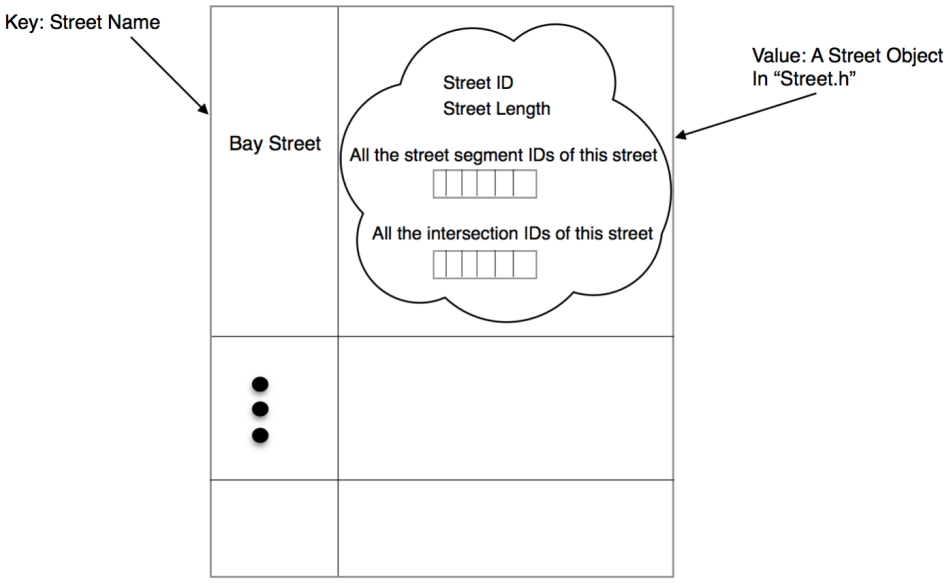

Figure 2: The above diagram is the unordered map "Hash_Street_Info". It contains information for all the streets. The keys are street names and the values are Street objects. The data members of Street are a street ID, a street length, a vector of all the street segment IDs of that street, and a vector of all the intersection IDs through which the street passes.

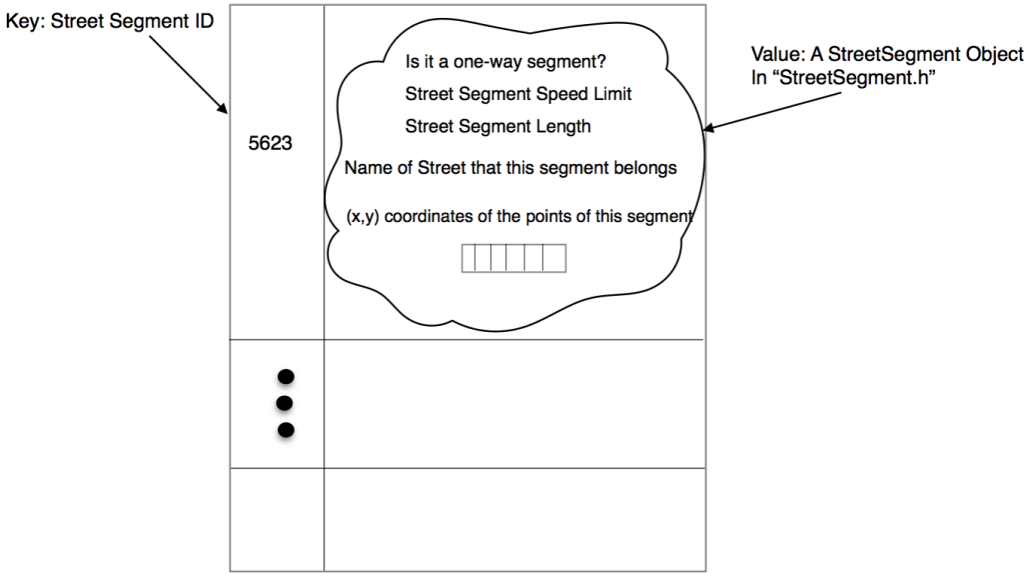

Figure 3: The above diagram is the unordered map "Hash_StreetSegment_Info". The keys are street segment IDs and the values are StreetSegment objects. The data members of a StreetSegment are if the segment is one way, the speed limit, the segment length, the name of the street this segment belongs to, and a vector of the (x,y) coordinates of the points of this segment.

All files other than m1.cpp, m2.cpp, m3.cpp, and m4.cpp are classes, similar to StreetSegment.h, and are stored in other data structures. An example is shown below.

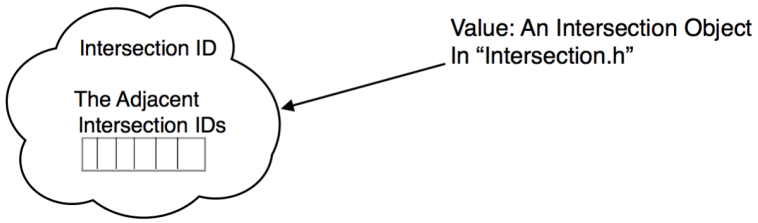

Figure 4: This diagram is an Intersection object in the file “Intersection.h”. It contains the Intersection ID and a vector of the IDs of its adjacent Intersections. This object is stored in Hash_Intersection_Info.

- **m2.cpp**

  This file handles the GUI and draws the map by using the information in our data structures loaded in m1.cpp. The most used data structure in this file is HashStreetSegmentInfo as it contains the coordinates of the segment points and the name of the streets. A graphics API is used to draw the map.

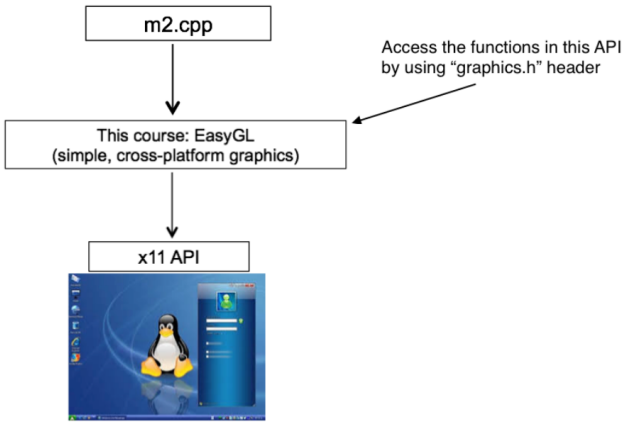

Figure 5: The above figure shows how to use the Graphics API. Functions such as setcolor(), drawline(), and create_button() are in the “graphics.h” header file. These functions are used in m2.cpp. [2]

M2 is also responsible for handling interactions with the GUI, such as clicking mouse buttons, moving the cursor, pressing keyboard keys, or zooming. For example, if the user searches for an intersection, m2.cpp will parse the name entered and check whether that intersection exists by looking it up in the data structures in m1.cpp.

```
event_loop(act_on_button_press, act_on_mouse_move, act_on_key_press, drawscreen);
```

Figure 6: The above line of code shows the function in m2.cpp responsible for handling events and interactions with the GUI. All the parameters passed into the function event_loop are function pointers. For example, act_on_key_press is a function handles keyboard presses by the user and reacts accordingly.

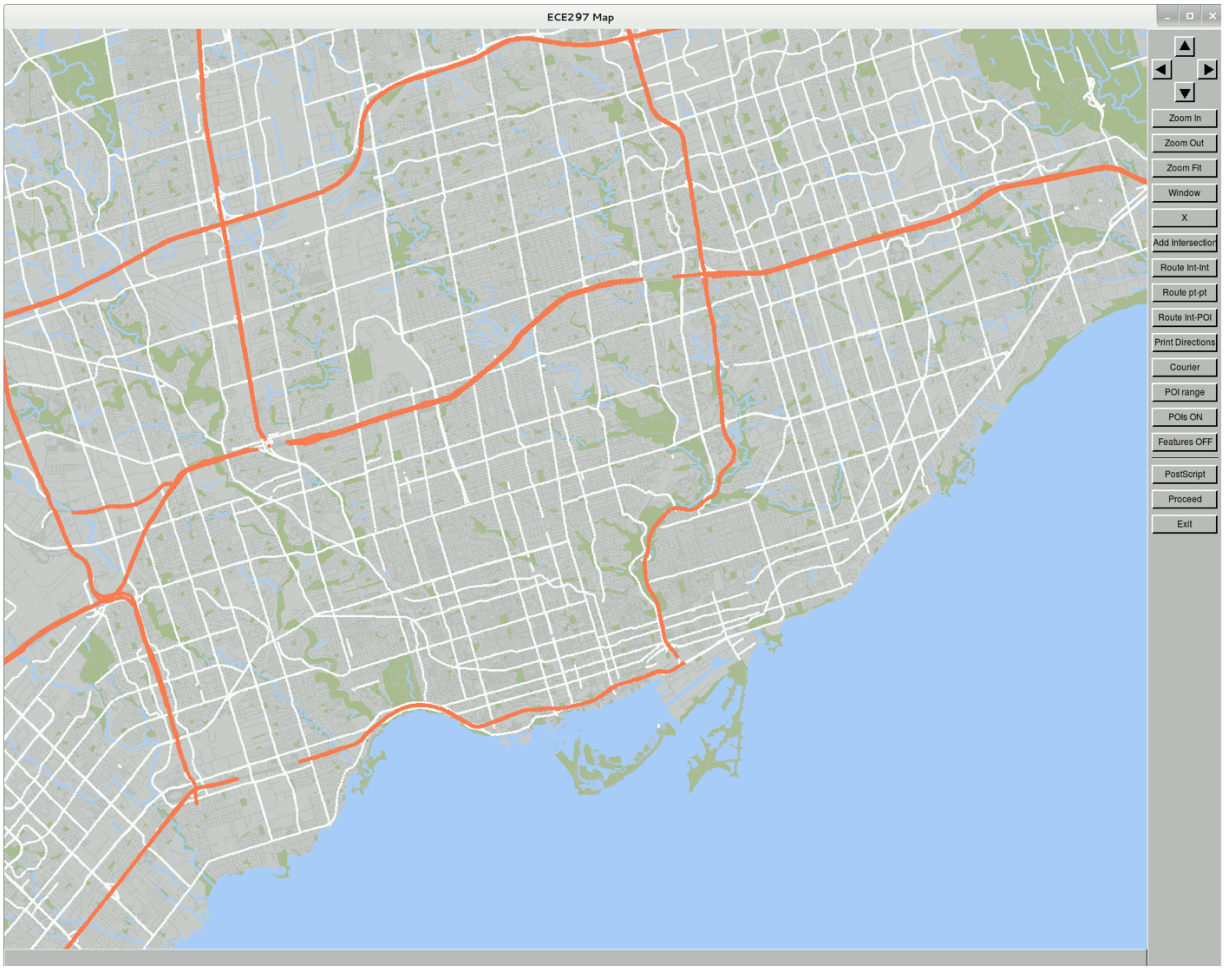

Figure 6: This figure shows the GUI created by m2.cpp and the list of buttons the user can interact with. Buttons include “Route pt-pt”, to find the path between any two points, or “Print Directions” to print textual navigations on the terminal.

- **m3.cpp**

  This file includes functions to find an optimal path between two points in a map.

- **m4.cpp**

  M4.cpp finds a path through multiple points. Its main purpose is to mimic softwares used by courier companies for a delivery truck to start at a depot, go to every single delivery location, and drop the truck back at any other depot. The algorithms and data structures used in m3 and m4 are explained in detail in the following sections.

- **main.cpp**

  This file prompts the user to enter a city to view the city map. The OSM data of that map is then loaded into our data structures and the map is drawn, allowing the user to interact with the GUI. If the user wants to find a path between points in the map, then m3.cpp and m4.cpp contain the functions responsible for it.

B. Major Algorithms Used

Many algorithms were used in this project, ranging from drawing street names to finding the fastest route between points.

- **Street Names**

  The text should be rotated by an angle so that it is correctly printed on the street segment. The positive angle is considered to be counterclockwise from the horizontal line.

  Straight Segment:

  We want to find the angle formed by the street segment and the horizontal line. After rotating the the name by θ it should appear as shown.

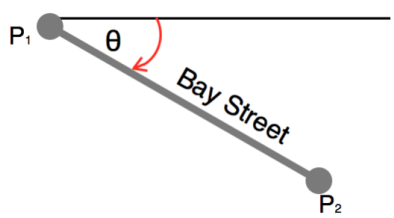

Figure 7: This figure shows the street segment and its endpoints in grey and the angle θ that the street name should be rotated by in order for it to be displayed correctly. θ is relative to the horizontal line.

$P_1 = (P_{1x}, P_{1y})$ $\quad \Delta Y = P_{2y} - P_{1y}$ $\quad \theta = \mathrm{Tan}^{-1}\left(\frac{\Delta Y}{\Delta X}\right) * \frac{180}{\pi}$

$P_2 = (P_{2x}, P_{2y})$ $\quad \Delta X = P_{2x} - P_{1x}$

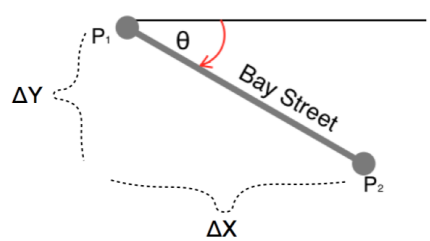

Figure 8: This figure shows the equation used to compute the angle θ that the street name should be rotated by. This equation computes the correct θ for any straight line segment in any quadrant.

Curved Segment:

A curved segment is composed of numerous straight lines. Now separate the name into individual characters and print each character on one of these red lines. The same formula is used to find θ. Previously, we rotated the whole name by θ whereas in this case we have to compute the θ for each red line to rotate each letter by that amount.

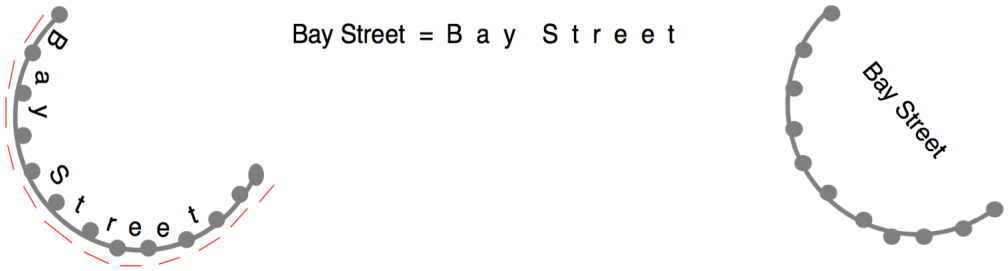

Figure 9: The left figure shows how the street name will appear for segments which are curved. The street name is broken up into its individual characters and each character is rotated by its own calculated θ (formula in Figure 8). If the straight line algorithm is used, the name for curved segments would appear incorrectly as shown on the right.

The following rule is used to decide which street names to draw. The drawtext() function in graphics.h takes two parameters which define a rectangle, boundx and boundy. If the name does not fit within the rectangle shown in Figure 10, then it is not drawn. This was done to avoid names appearing at a far zoom level and thus preventing clutter. This is also an advantage as it does not require checking the zoom level to decide whether the names should be displayed or not.

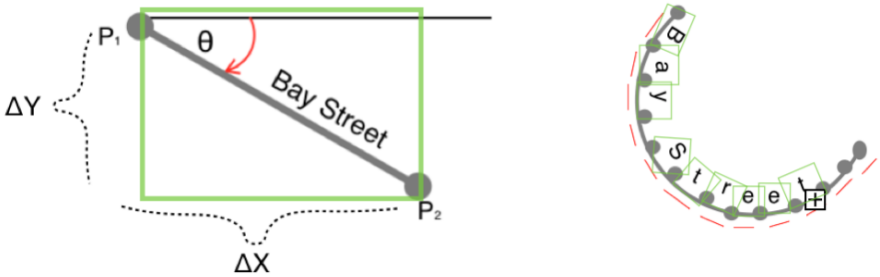

Figure 10: This figure shows when the street names are drawn. It is independent of the zoom level. Parameters are passed into the function drawtext() to only draw the street name or individual characters (in the case of curved segments) if the text fits inside the green rectangle.

Figure 11: This figure shows how street names appear currently on our user interface.

- **Path-Finding**

Breadth First Search

The data structures are shown in Figure 12. Start at the initial Intersection and spread outwards in circles. We have a “WaveFront” list which contains “WaveElement” objects representing the adjacent Intersection nodes. Each WaveElement in the WaveFront has a pointer to the node to be inspected, the segment ID used to reach that node, and the travel time from the source to that node. For every node, we add a WaveElement to the WaveFront for each of its adjacent intersection nodes to be inspected.

We inspect each of those adjacent intersections and add their respective adjacent intersections to the WaveFront as well. During the inspection of each node, we store the shortest travel time from the source to this node and mark this node as visited. A node is only allowed to be revisited if the shortest travel to it has dropped (ie. The time in the WaveElement is < The shortest time stored at the node). Revisiting includes updating the shortest travel time and the segment ID used to reach this node.

If the WaveElement in the WaveFront ends up representing the destination node, then a path to the destination exists. However, keep on inspecting WaveElements in the WaveFront until all entries in the WaveFront have a higher travel time than the best path found so far to make sure we have found the fastest route.

BackTrace the Path

We start at the destination and see which edge was used to reach this node. We keep backtracking until we hit the initial node. During this process, we store the segment IDs in a vector to construct the path.

Dijkstra’s Algorithm

Instead of visiting the WaveElements in order, inspect the smallest one in the WaveFront first (ie. The entry with the shortest travel time from the source to this node).

Heap

The WaveFront is a list, thus it can take O(N) to find the smallest WaveElement as the entire list should be scanned. Use a heap to keep the smallest WaveElement on top, allowing us to have an insertion and removal complexity of O(logN). Tweak the comparator to change it to a minimum heap rather than the default maximum heap.

A*

Estimate the minimum time from any node to the destination. This heuristic is a lower bound and is lower than the true time of moving from a node to the destination. This is the fastest time you can get to the destination from any node (ie. Driving on a highway at the maximum speed limit). Using this heuristic, we can incorporate it into the WaveFront sorting, such that the minimum element of the heap is the most promising solution.

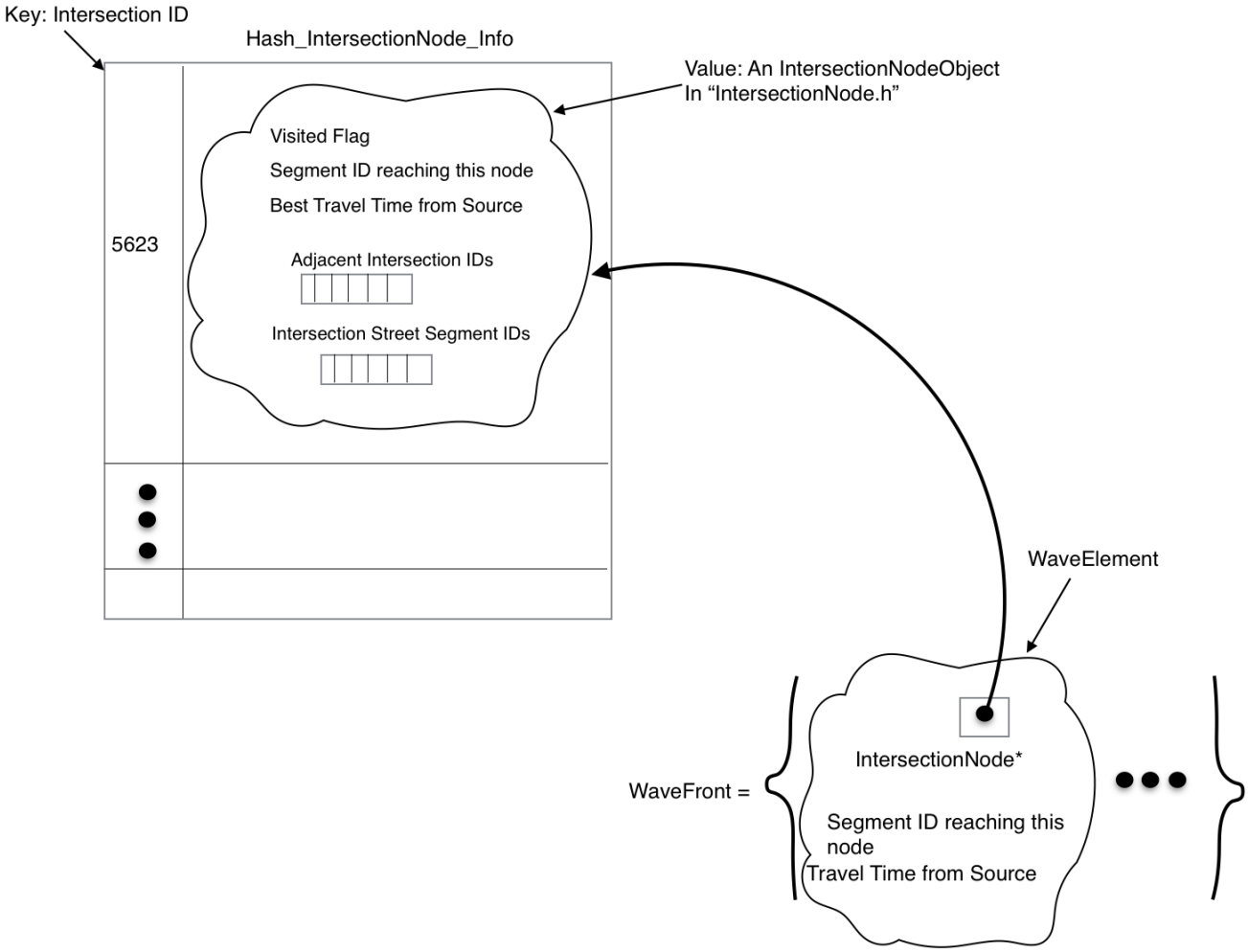

Figure 12: Figure showing the data structures used in the path finding algorithm. The WaveFront is a minimum heap. The WaveElement has pointers to the Intersection nodes that should be inspected next and that are most promising solutions. The WaveElement also has the segment ID used to reach this node (for BackTracing) and the travel time from the source to this node. Intersection nodes are allowed to be re-visited only if the travel time in the WaveElement is less than the travel time in the “IntersectionNode” in “Hash_IntersectionNode_Info”

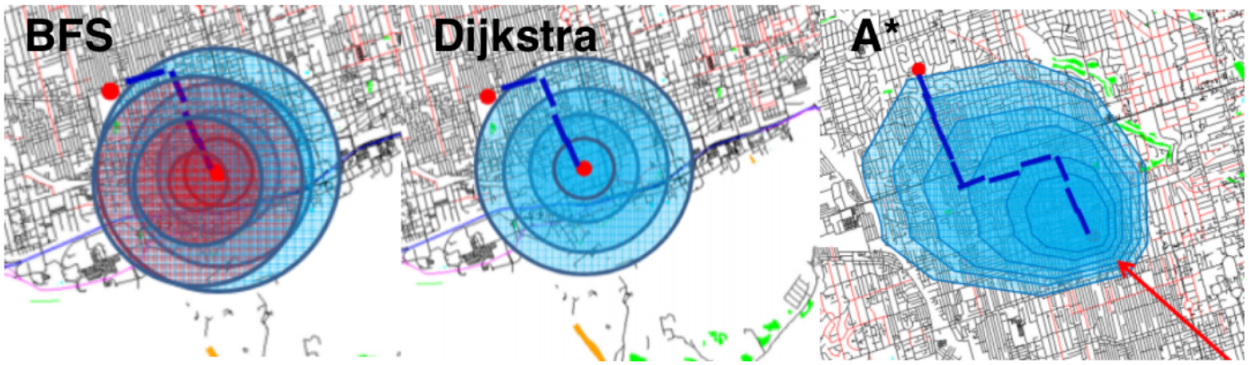

Figure 13: Figure showing the difference between the 3 path finding algorithms. Clearly A* on the right is the best. [3]

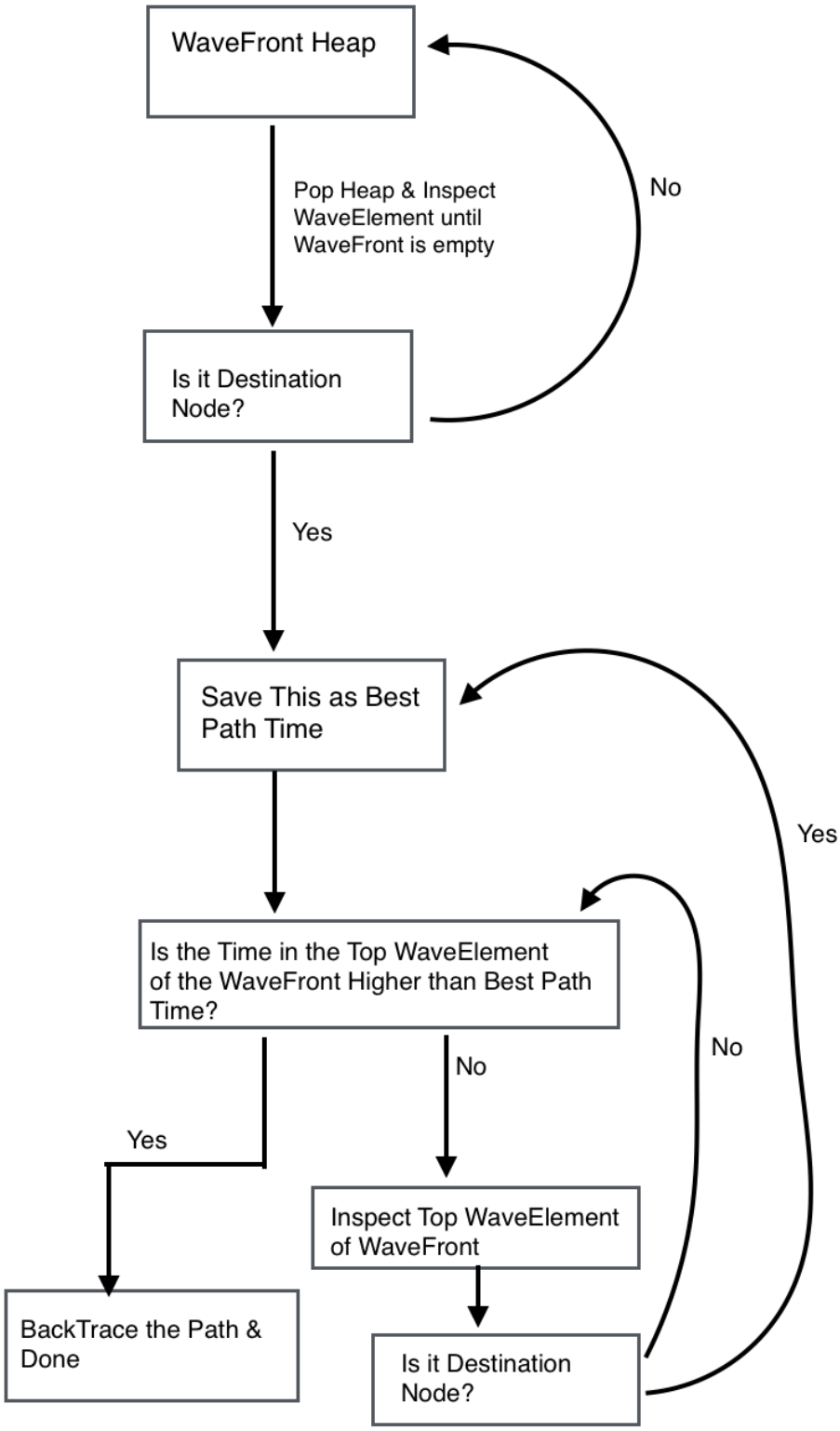

Figure 14: Flow Chart showing how the path finding algorithm works by inspected WaveElements in the WaveFront Heap.

Algorithm used	Time to Pass the Hardest Test Case in ece297exercise
BFS with re-visiting nodes	28 minutes
Dijsktra's (with the WaveFront being a minimum heap)	22.5 seconds
A* (Best First Search)	4 seconds

Figure 14: Figure showing the time each algorithm took to pass the hardest test cases in ece297exercise regarding finding a path between two points. Again, clearly A* has the fastest response time.

- **Courier Company Route Optimizer**

  Start by precomputing the path from each delivery ID to the every other delivery ID. This is Dijkstra's algorithm to have a single source and multiple destinations. This results in having the path and time from each node to every node. The pre-computation time for 50 delivery IDs is 6.12 seconds. This means each Dijkstra call takes 0.12 seconds which is really fast.

  The basic path is found using greedy algorithm. We start at a delivery ID and go the next closest delivery and keep on repeating that until we visited all the delivery IDs. This results in a QoR of 240. Up to this point, we have used 6.43 seconds of the 30 second time-limit, leaving us with 24 seconds of optimization time.

  The order of the delivery IDs are stored in a vector. We have to optimize this delivery list to achieve a better path.

  1. **2-Point Swap**

     Pick two random indexes in the vector of delivery IDs and swap their position. This allows for the exploration of more space randomly.

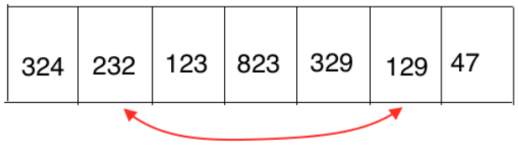

Figure 15: This figure shows how the 2-point swap algorithm works. It picks two random numbers in the vector and swaps their position in the vector of delivery IDs. In this case 232 and 129 are swapped.

This always finds at least one better solution than the path found by greedy algorithm. The swapping is done via swap in <algorithm> and has a complexity of O(N). Around 4,000,000 swaps are done.

2. Local Perturbations

Local perturbations are applied to improve the path locally. This involves picking two points in a vector and reversing the piece in between those points. Every possible pair of points in the vector of delivery IDs should be considered. This is (N-1) choose 2 choices, which equals 1176 if N=50 deliveries. If we find an improvement, the function is called again as a perturbation that didn't work before might work now. An example of reversing a piece is shown in Figure 17.

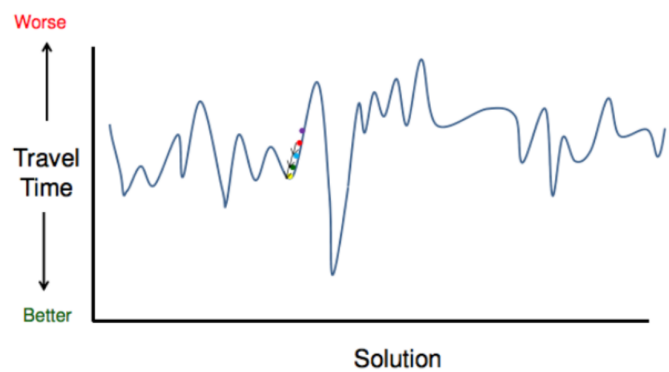

Figure 16: Graph showing the space of possible solutions. Iterative improvements and local perturbations result in being stuck in a local minimum and not being able to find better solutions. [3]

3. 2-Opt

This causes larger perturbations and can explore bigger spaces and escape local minima. Two random indexes are chosen and the path is cut into 3 pieces. Now the path must be reconnected differently in order for it to be legal. Methods such as reversing the second piece, or rotating piece 2 and 3, or even both are used. Reversing and rotating are done by functions in <algorithm> and both have complexity of up to O(N), N being the number of delivery IDs

```
          deliveryOrder = {0, 1, 2,|3, 4, 5, 6,| 7, 8, 9}
Reverse   deliveryOrder = {0, 1, 2,|6, 5, 4, 3,| 7, 8, 9}
Rotate    deliveryOrder = {0, 1, 2,|7, 8, 9,|6, 5, 4, 3}
```

Figure 17: This figure shows how a 2-opt works. Two random indexes separate the vector into 3 pieces. These pieces are reversed and rotated to connect the delivery IDs differently but legally. In this figure, we first reverse the second piece and then rotate pieces 2 and 3.

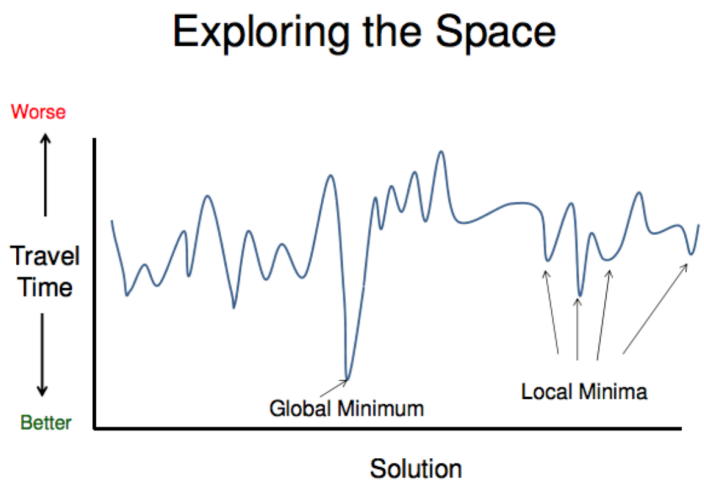

Figure 18: Graph showing the space of possible solutions. Major perturbations, such as 2-opt and 3-opt, followed by local perturbations might allow us to get close to a global minimum. [3]

For all the algorithms, a new solution is accepted only if the new travel time is better than the best stored. Once we have optimized our delivery IDs, we add the Depot IDs to it. We find the closest depot to the first Delivery ID and the closest depot to the last one. These algorithms result in a QoR of 196 minutes, which is better than the 240 minutes found by the Greedy algorithm.

C. Data Structure Choices

Many of the data structures used are unordered maps in order to maintain a read and insertion complexity of O(1). For example, during the pre-computation of the path between the delivery IDs, an unordered map was placed inside another unordered map to store the path and the time.

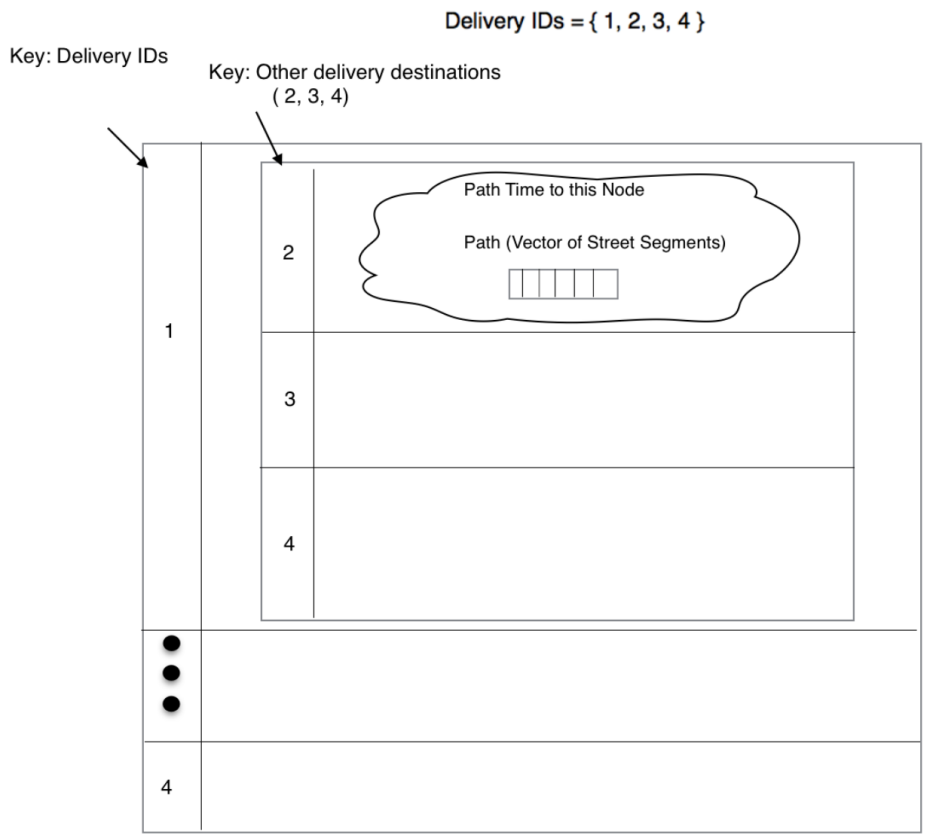

Figure 19: Figure showing the data structure used to store the path and time from each delivery node to every other delivery node during the path pre-computation. It is an unordered map stored inside an other unordered map. The object in the inner unordered map contains the path to this node and the exact time to this node.

The minimum heap used in Dijkstra's algorithm also increased the performance. Previously, the WaveFront was a list and finding the minimum WaveElement had a complexity of O(N). A minimum heap always has the smallest element on top and has an insertion and removal complexity of O(logN). The IntersectionNode pointer in the WaveElement (Figure 12) also avoided making copies of IntersectionNode objects.

Such data structures helped immensely in achieving a pre-computation time of 6.12 seconds which is excellent.

3. Product Overview

Many features were added to our final design to reinforce our main goals.

- **Path Finding Tool**

  Navigation systems are mainly used for time-saving purposes and to travel from one point to another using the fastest route available. Our goal was for our path finding features to be used both personally and commercially in order to target a larger number of users. Therefore, we implemented variations of our path finding features which allow users to find an optimal path through either multiple points or two points. A commercial application of this feature is its use by courier companies to get an order and route for a set of deliveries, for an effective delivery schedule. Alternatively, a personal use of this feature is a tourist wanting to find a route through multiple destinations in a city. The path finding features available in our interface is shown below.

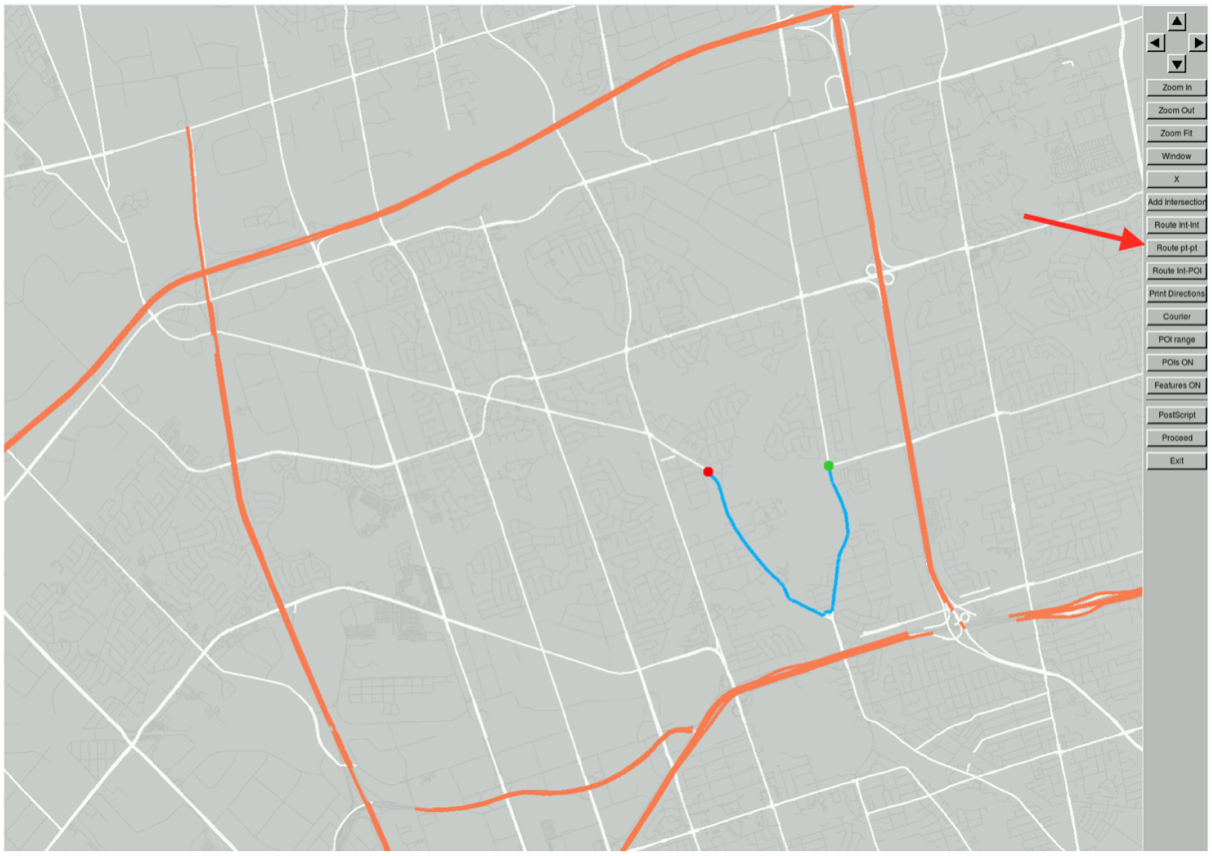

Figure 20: Figure showing the usage of our path finding feature between two points. This feature is used by clicking two points on the map and upon clicking the “Route pt-pt” button (shown by the red arrow), a path is shown between the points. The starting point is shown in green and the destination is shown in red.

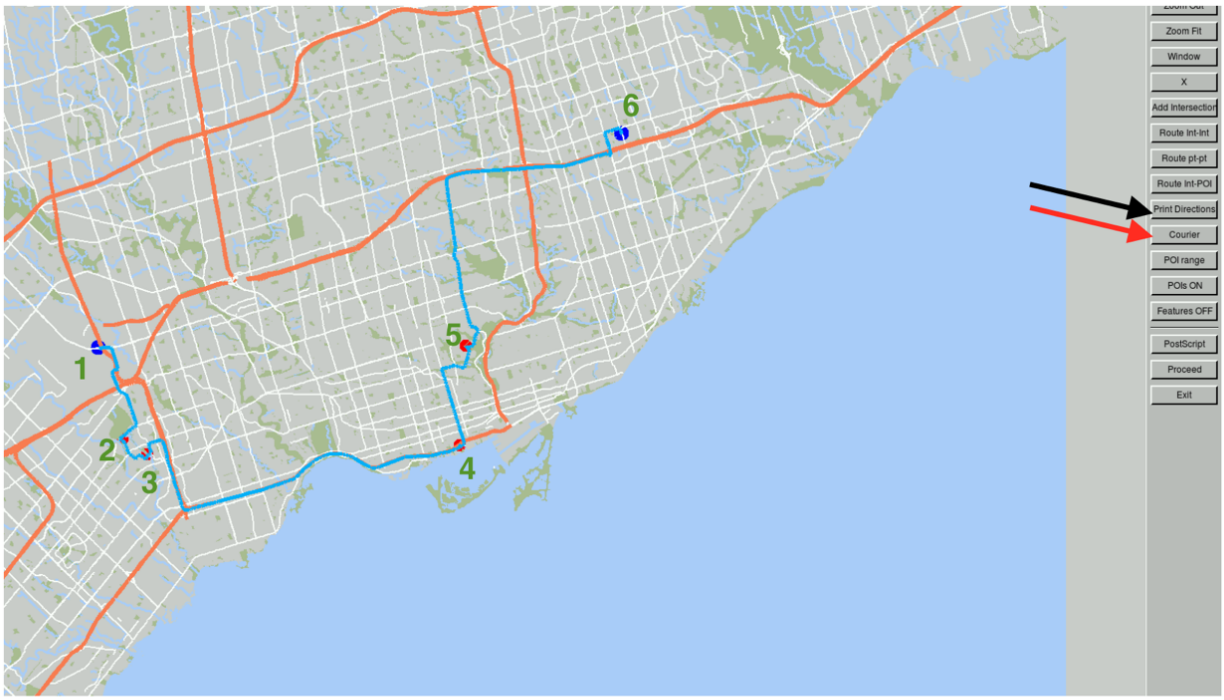

Figure 21: Figure showing the usage of our path finding feature through multiple points. This feature can be used commercially (by UPS) and personally (by tourists) to find an order and route through multiple destinations. Multiple points are clicked on the map and upon clicking the “Courier” button (shown by the red arrow), a path is shown through the points. The order is also numbered in green on the map. The black arrow shows the “Print Directions” button.

- **Meeting Personal Preferences**

  Since our goal is to target a wider number of users by creating variations of path finding features, we have to also meet various personal preferences. Some prefer having a visual representation of the path, while others prefer to see navigational directions printed out. As a result, a button called “Print Directions” was added. This feature is shown using the black arrow in Figure 21 and the result of pressing it is shown below.

```
----------------DIRECTIONS ----------------
Starting From College Street (44)
Start on College Street
Stay on College Street for 9 metres.
Turn on University Avenue
Stay on University Avenue for 1122 metres.
Turn on Richmond Street West
Stay on Richmond Street West for 53 metres.
Turn on Simcoe Street
Stay on Simcoe Street for 9 metres.
Reached Destination: Simcoe Street (17)
-------------------------------------------
```

Figure 22: Figure showing the output of pressing the “Print Directions” button indicated by the black arrow in Figure 21. Textual navigation of the path found is printed out.

Another discrepancy in preferences is the method by which features are used. Some prefer to click on points and find a route through them, while tourists prefer to enter destination names as they are unfamiliar with the city. To account for that, we have implemented a search bar which allows the user to enter destination names as well. Since mistakes often happen during typing, we have implemented an autocorrect and an autocomplete feature in the search bar. An autocorrect feature allows the user to enter both “Bay & College” and “College & Bay”.

Figure 23: Figure showing the search bar (shown by the red arrow) on the bottom of the user interface. Any destination name (Tim Hortons) or intersection name (Bay & College) can be entered. After entering the desired number of intersections, the button “Route Int-Int” will display a path through these points. This buttons is shown by the black arrow. The search bar is equipped with an autocomplete and autocorrect feature.

- **Fast Response-Times**

The response times of key tasks such as redrawing the map and finding a path were measured to ensure the interface's quality. A summary of the results are shown in the Figure below. By analyzing the results, one can conclude that the time taken for executing key tasks will be almost unnoticeable while using the interface.

Key Tasks	Response Time	Conclusion
Redrawing the Screen	0.02 - 0.03 seconds	Redrawing the screen as a result of zooming or panning is almost unnoticeable.
Finding a Path Between Two Points	0.04 - 0.08 seconds	If two close points are chosen, the response time is 0.04 seconds. On the other hand, if the 2 furthest points on the map is chosen it takes 0.08 seconds. This is an excellent response time for the path finding function.
Printing Directions	0 seconds	Prints out the results instantaneously.
Generating Route for Courier Company	8 - 24 seconds	It takes 8 seconds to generate an average path but it takes 24 seconds to optimize the path resulting in an optimal path through the points. Since the time limit set by the instructors is 30 seconds, any time under 30 seconds is acceptable.

Figure 24: Figure showing the response times of key tasks of the interface such as redrawing the screen, finding the path between 2 points, printing directions, and generating a route for a courier company. By looking at the response times, one can conclude that they are extremely fast for interactive use and almost unnoticeable.

List of References

[1] Vaughn Betz, Winter 2015, *"Administrative Stuff"* [Online]. Accessed: April 5th, 2015. Available: http://www.eecg.toronto.edu/~vaughn/ece297/lectures/lecture3_osm_and_graphs_betz.pdf

[2] Vaughn Betz, Winter 2015, *"Good Coding Style and Graphics APIs"* [Online]. Accessed: April 5th, 2015. Available: http://www.eecg.toronto.edu/~vaughn/ece297/lectures/lecture7_code_style_and_graphics_betz.pdf

[3] Vaughn Betz, Winter 2015, *"Heuristic 4: Hill Climbing"* [Online]. Accessed: April 6th, 2015. Available: http://www.eecg.toronto.edu/~vaughn/ece297/lectures/lecture20_traveling_salesman3_betz.pdf

[4] Médecins Sans Frontières, July 7th, 2014. *"GIS Support for the MSF Ebola Response in Guinea 2014"* [Online]. Accessed: March 30, 2015. Available: http://cartong.org/sites/cartong/files/GIS%20Support%20for%20the%20MSF%20Ebola%20Response%20in%20Guinea_Case%20Study.pdf

[5] R. A. Stein, *"What is EBOLA?*, in The International Journal of Clinical Practice, 2014 © Int J Clin Pract. doi: 10.1111/ijcp.12593

[6] American Red Cross, August 5th, 2014, *"Volunteers Around the World Aid Ebola Victims"* [Online]. Accessed: April 1st, 2015. Available: http://www.redcross.org/news/article/Volunteers-Around-the-World-Aid-Ebola-Victims

[7] T.S. Key, 2011, *"Future of Renewable Energy Development & Deployment"* [Online]. Accessed: April 2nd, 2015. Available: http://journals1.scholarsportal.info.myaccess.library.utoronto.ca/pdf/19449925/v2011inone/1_foredd.xml

[8] MapBox, 2015, *"A Data Platform"* [Online]. Accessed: April 2nd, 2015. Available: https://www.mapbox.com/data-platform/

[9] Humanitarian Openstreetmap Team, 2015, *"Get Involved"* [Online]. Accessed: April 4th, 2015: Available: http://hot.openstreetmap.org/get-involved

[10] Esri, 2015, *"ZKI Activations"* [Online]. Accessed: April 4th, 2015. Available: http://www.zki.dlr.de/sites/all/libraries/jsviewer/JSActivation_Map.html#style=large#lang=en

[11] American Red Cross, Spring 2014, *"West African Ebola Response"* [Online]. Accessed: April 4th, 2015. Available: http://americanredcross.github.io/guinea-swipe/?americanredcross.fy5019k9&osm/#158.5641/-10.1361

[12] OpenEI, 2015, *"Solar and Wind Energy Resource Assessment (SWERA)"* [Online]. Accessed: April 5th, 2015. Available: http://en.openei.org/wiki/SWERA/Data

[13] OpenEI, 2015, *"Welcome to OpenEI: Energy Information Database"* [Online]. Accessed: April 5th, 2015. Available: http://en.openei.org/datasets/dataset

[14] National Renewable Energy Laboratory, 2015, *"Solar and Wind Energy Resource Assessment"* [Online]. Accessed: April 5th, 2015. Available: http://maps.nrel.gov/swera

[15] International Renewable Energy Agency, 2015, *"Global Atlas for Renewable Energy"* [Online]. Accessed: April 6th, 2015. Available: http://irena.masdar.ac.ae/

[16] Bioenergy Knowledge Discovery Framework, 2015, *"Bioenergy KDF REST API"* [Online]. Accessed: April 7th, 2015. Available: https://bioenergykdf.net/apidocs/restapi