Incident Management System
For Intelligent City Development

(Group 2)

INDIVIDUAL FINAL REPORT
(Awareness of Traffic and Incidents, Backend Development)
14th April 2019

fyp18012
Chow Yin Chak, Terence
3035271529

Supervised by Dr. Cheng, Reynold C.K.
Abstract

This project aims to provide a solution to alleviate the traffic congestion problem in Hong Kong. A traffic incident- and traffic usage-aware routing system is implemented based on a self-proposed road network preprocessing method. Such method speeds up heuristics calculation in real-time queries with a layered precomputation approach. It prevents expensive recomputation of preprocessed data upon graph updates and enables sensitivity of traffic density on the roads at different time points. The final system includes a backend and a frontend application serving the purpose of experiments and demonstration of this project. The system was found to provide up to 22% improvement in average travelling time and more than 85% of requests can be served with a path that has shorter travelling time than routing without traffic and incident awareness.
Acknowledgments

I would like to express my whole-hearted gratitude to the project supervisor, Dr. Reynold Cheng, for guiding the team throughout the project.
# Contents

Abstract ...................................................................................................................................................................... 1  
Acknowledgments .................................................................................................................................................... 2  
Contents ..................................................................................................................................................................... 3  
List of Figures ........................................................................................................................................................... 4  
List of Tables ............................................................................................................................................................. 4  
1 Introduction ..................................................................................................................................................... 5  
   1.1 Background and Motivation ................................................................................................................... 5  
   1.2 Objective and Scope ................................................................................................................................ 6  
   1.3 Outline of Report ..................................................................................................................................... 7  
2 Methodology .................................................................................................................................................... 8  
   2.1 Awareness of Traffic Information ......................................................................................................... 8  
      2.1.1 Traffic Density ................................................................................................................................. 9  
   2.2 Time-independent Incidents ............................................................................................................. 13  
      2.2.1 Technologies ................................................................................................................................... 15  
      2.2.2 System Classes and Components ................................................................................................ 16  
   2.3 Function Design ..................................................................................................................................... 23  
      2.3.1 Routing Service .............................................................................................................................. 23  
      2.3.2 Reroute Service .............................................................................................................................. 24  
      2.3.3 Incident Injection ........................................................................................................................... 27  
      2.3.4 Incident Removal ........................................................................................................................... 28  
3 Results and Discussions ................................................................................................................................ 29  
   3.1 Traffic Awareness .................................................................................................................................. 29  
      3.1.1 50 Requests Set .............................................................................................................................. 30  
      3.1.2 500 Requests Set ............................................................................................................................ 31  
      3.1.3 1000 Requests Set .......................................................................................................................... 33  
      3.1.4 Summary ......................................................................................................................................... 34  
   3.2 Incident Awareness ................................................................................................................................ 35  
   3.3 Combined Awareness ............................................................................................................................ 36  
4 Conclusions .................................................................................................................................................... 37  
5 References ....................................................................................................................................................... 39
List of Figures

Figure 1: Illustration of Edge Effective Area ................................................................................................. 14
Figure 2: Software Architecture of Backend .................................................................................................. 16
Figure 3: Screen Capture of Graph Builder .................................................................................................... 22
Figure 4: Routing service backend sequence diagram .................................................................................. 23
Figure 5: Reroute service backend sequence diagram .................................................................................. 25
Figure 6: Incident Injection handling backend sequence diagram ................................................................. 27
Figure 7: Incident Removal handling backend sequence diagram ................................................................. 28
Figure 8: Reported Travelling Time of Each Request in 50 Request Set ...................................................... 30
Figure 9: Reported Travelling Time of Each Request in 500 Request Set ..................................................... 31
Figure 10: Spikes-free Travelling Time for 500 Requests Set ...................................................................... 32
Figure 11: Reported Travelling Time of Each Request in 1000 Request Set ................................................... 33
Figure 12: Spikes-free Travelling Time for 500 Requests Set ...................................................................... 33
Figure 13: Routes with and without existence of incidents ........................................................................... 35
Figure 14: Original Paths and Alternative Paths ............................................................................................. 36

List of Tables

Table 1: Data Stored in MapGraph .................................................................................................................. 17
Table 2: Data and Methods in IncidentManager ............................................................................................ 19
Table 3: API Endpoints available in IMSApp .................................................................................................. 20
Table 4: Aggregated Results of 50 Request Set .............................................................................................. 30
Table 5: Aggregated Results of 500 Request Set ........................................................................................... 31
Table 6: Aggregated Results of 1000 Request Set .......................................................................................... 33
Table 7: Travelling Time of Paths with and without awareness of incidents .................................................. 35
1 Introduction

1.1 Background and Motivation

Road incidents and large-scale events has been the main reasons for severe traffic congestions. This leads to huge waste of valuable time and even economic loss. In fact, citizens in Hong Kong may suffer at least a 200% increase in commuting time in serious traffic jams [1]. Therefore, a travelling route suggestion service that can take incidents into consideration may provide a way to alleviate the problem. Currently, amongst all available solutions for this problem, TallyGo [2] (former ClearPath) provides the most similar service. It detects and takes live incidents and road congestions into consideration and suggests alternative paths to the user to ensure reasonable travelling time and prevent worsening the congestion. Despite being a well-established product, there are still shortcomings of TallyGo that prevents it from being directly used in Hong Kong:

Firstly, large-volume requests are not specifically handled in TallyGo. In such cases, a gigantic amount of O.D. requests with the same origin may be sent out at the same time when many users are trying to leave an event. If all requests are served with the same suggested route, new traffic congestion may be created by the users on same path segments, the efficiency of the application may then be diminished. Secondly, TallyGo does not support Hong Kong as one of its service regions for routing with incident data because of the lack of accurate incident and traffic data available in Hong Kong.

Serving O.D. requests can be modeled as a shortest path problem applied to a road network. The road network can be modeled as a graph with positive weights. The nodes represent road junctions and each edge associated with a road segment between two junctions.

This shortest path finding problem is in fact very popular and well-studied. Throughout the researches, popular algorithms like Contraction Hierarchies (CH) [3] and Customizable Contraction Hierarchies (CCH) [4] have been presented for speeding up real-time queries. These algorithms work similarly. The graph data are first preprocessed without the knowledge of origin and destination. Shortest path requests can then be served with given origins and destinations. These algorithms assume a static road network, such that the network only needs to be preprocessed once and can be used for a large number of O.D. requests.

However, in many use cases the road network is not static, 2 main scenarios can cause the changes.

1. Traffic incidents and events potentially affect the travelling of road segments.
2. A lot of other vehicles are served and routed in the road network, causing substantial traffic in the road network.

The algorithms mentioned previously lost their advantages in these scenarios. In CH, the speed up strategy used includes initial edge weights (default travelling time) which are changed or lengthened in these case scenarios in their precomputed data and relies on these values to find short-cuts to accelerate the search. As a result, time-consuming computation on the graph data has to be done repeatedly whenever there are changes in the road network. On the other hand, in order to be aware of the other traffic usage and yield more useful routing results, time-sensitivity and graph updates are required in the edge weights so that
existence of dynamically added incidents and future traffic densities predicted with routes of the other traffic in the system can be incorporated in the path finding procedures. Even in CCH where customization of edge information can be done quickly, time-sensitivity still cannot be achieved as each customization only reflects information at one moment. Such customization needs to be done very frequently to capture all the changes in traffic densities this path will yield for each involved path at each time point. These methods therefore may not be feasible in a real-time routing application with dynamic map information like road edge weights.

1.2 Objective and Scope

We hypothesize that there is significant value to get a route planning service to detect and react to real-time traffic incidents autonomously. Due to the wide breadth of the two components and time constraints, this project focuses on how to take real-time events into consideration while doing shortest path searches.

Understanding the limitations mentioned above, this project also delivers a web-based incident management service (IMS) focusing on Hong Kong data featuring a proposed set of algorithms in which O.D. requests are served real-time with requests that are not be known by the system beforehand. The system ensures efficiency of the routing algorithm to maintain user experience and finds the quickest path with traffic information like existence of incidents and traffic density of the roads.

There are 3 major deliverables in this project:

1. **Routing Algorithm**
   A set of algorithms designed to solve origin to destination routing requests that take real-time events into consideration. The proposed method features a *preprocessing* phase which performs a layered preprocessing of map data; a *query* phase which makes use of the common A* searching algorithm with a realized and a future cost function, followed by an *update* phase in which the searched path is fed back to the system to update the traffic density of each edge involved in the path at each time point the vehicle enters the edge. Traffic incidents can be fed into the algorithm at any time by their location and impact on the travelling time.

2. **Backend service**
   A set of functions implemented as web API endpoints to provide routing, rerouting and incident management functionalities to the Frontend service. Acts as the controller to handle request parameters (e.g. origin and destination location in routing) and pass them to corresponding components for the results needed to be returned to the Frontend.

3. **Frontend service**
   A mobile application acts as both application prototype and simulator.
   
   a. Application prototype provides a route planning service. It also includes basic features from ordinary mapping mobile application.

   b. Simulator simulates an environment which involves multiple users in the road network, mimicking the real world. The environment can be used to test and improve the routing algorithm efficiently and quickly.
1.3 Outline of Report

The details of the preprocessing and routing algorithms, as well as the frontend implementation will be discussed in separate reports of this FYP group. This report will mainly focus on the application of the proposed path finding method, including the incorporation of awareness of traffic and incidents (the update phase), as well as the workflow of the final backend service. This report first describes the methodologies including the awareness of traffic and incidents in the system, backend technologies involved and the workflow of different system functionalities. Experiments for the performance of the final system will then be unfolded, followed by the final discussions and conclusions.
2 Methodology

2.1 Awareness of Traffic Information

A useful routing result for the users should be the of the shortest travelling time according to the current traffic information. In the scope of this project, such information includes the traffic density, which is the number of cars per unit length of the road, and the existence of incidents that may lengthen the travelling time of the affected roads. A feasible route returned therefore should be able to avoid involving roads that are congested by other vehicles and those affected by any traffic incidents so that it has shorter total travelling time.

In this project, the common A* path finding algorithm is used for searching for the quickest path to serve the O.D. requests. While the data generated in the preprocessing stage are used as the heuristics to be used in A* computing the expected lower bound of travelling time (This is always admissible as the realized travelling time will never be shorter than the default value, an under-estimation of the default travelling time will therefore always be shorter than the realized travelling time. Please see the report of another groupmate for a detailed proof), the information mentioned above are used as the major contributing factors in the realized cost function $g(s, u)$ of A*:

$$g(s, u) = \begin{cases} 0, & \text{if } s = u \\ (g(s, u') + w' ((u', u), t_0 + g(s, u'))), & \text{otherwise} \end{cases}$$

Where the weight function of an edge $w'$ can be expressed as the sum of realized travelling time (calculated by function $rt$) and a time independent function $a$ representing the impact of traffic incidents as follows:

$$w'(e, t) = rt(t) + a(e)$$

By minimizing the realized cost, the final search result of the A* algorithm will be having the least sum of realized travelling time calculated with the traffic density and incidents. Ultimately, this can prevent the routing algorithm from routing to overly popular roads that are used by many other paths or those heavily affected / congested by traffic incidents.
2.1.1 Traffic Density

In this section, the relationship between traffic density and travelling time of a road will first be investigated. The details of implementing such relationship such that predicted density information at different time points can be incorporated into the routing procedures will then be discussed.

2.1.1.1 Relationship between Traffic Density and Travelling Time

By Greenshield’s Model in traffic flow theory, assuming an uninterrupted traffic flow, there exists a linear relationship between the traffic density and the travelling speed \[ \frac{v}{d} \] assuming an uninterrupted traffic flow, there exists a linear relationship between the traffic density and the travelling speed \[ \frac{v}{d} \]. This model is widely used in the macroscopic traffic analytics field for it being simple to calculate yet still maintaining fairly high accuracy. The model suggests the following relationship, where \( v \) = speed, \( v_{\text{max}} \) = maximum speed, \( d \) = traffic density and \( d_{\text{max}} \) = maximum traffic density:

\[
v = -\frac{v_{\text{max}}}{d_{\text{max}}} d + v_{\text{max}} \quad (1)
\]

Since the criteria of the “shortest” path problem this project is trying to solve refers to shortest travelling time, it would be sensible that the realized travelling time of a particular road can be found given its current traffic density. In order to achieve this, the equation can be further arranged as follows:

From (1):

\[
v = v_{\text{max}} \left(1 - \frac{d}{d_{\text{max}}} \right) \quad (2)
\]

As travelling time = distance / speed, default travelling time can be expressed as follows, let \( t \) = travelling time, \( t_{\text{min}} \) = default travelling time and \( g \) = geographical length of the road:

\[
t = \frac{g}{v} \quad (3)
\]

\[
t_{\text{min}} = \frac{g}{v_{\text{max}}} \quad (4)
\]

Substitute equation (2) into (3) yields:

\[
t = \frac{g}{v_{\text{max}} \left(1 - \frac{d}{d_{\text{max}}} \right)} = \frac{g}{v_{\text{max}}} \cdot \frac{1}{\left(1 - \frac{d}{d_{\text{max}}} \right)} \quad (5)
\]

Finally, substitute equation (4) into (5) gives:

\[
t = \frac{t_{\text{min}}}{\left(1 - \frac{d}{d_{\text{max}}} \right)} \quad (6)
\]

Now the current travelling time can be found with the default travelling time, the maximum traffic density and the current traffic density of the road according to equation (6). The value of default travelling time can easily be calculated as shown in equation (3) with the actual road length from the map data and the corresponding legal maximum speed. For the maximum traffic density, as there is not enough data available for obtaining such information for each road in Hong Kong, such value is estimated with method described in the followings with the normal sized vehicle length \( l \) of 5m (derived from the standard parking
space governed by the Planning Department [6]). Let $n_{\text{max}}$ = maximum number of vehicles on the road, this value can be obtained when the road segment is completely jammed such that there is no space between each vehicle. As traffic density = number of vehicles per unit length of a road, maximum traffic density can be express as follows:

$$d_{\text{max}} = \frac{n_{\text{max}}}{g} \quad (7)$$

For a completely jammed road, the spacing between each vehicle should be 0. The number of vehicles on the road will then be road length divided by car length:

$$d_{\text{max}} = \frac{g}{l} \quad (8)$$

As shown in equation (8), the maximum traffic density for all roads can be deduced as a constant value of $\frac{1}{5}$ regardless of the geographical length. As a result, the only variable that the current travelling time depends on becomes the current traffic density of the road, i.e. how many other vehicles are travelling on 1 unit length of this road at the current time point.

Equation (6) can now be used as function $rt$ in $w'$ to calculate the realized time needed to complete travelling on one road based on the traffic density of that road at each time point. This is particularly useful in the $A^*$ routing algorithm used in this project as a major part of the realized cost function. The implementation details of such relationship will be unfolded in the next section.
2.1.2  Current Density Updated with Returned Edges

According to equation (6), in order to obtain the predicted realized time needed to travel a road, the traffic density of this road at the enter time, which is the time when the vehicle is projected to enter an edge, is needed. Such traffic density information of each edge at different time points must therefore be recorded to enable the prediction in the routing. In the scope of this project, only the existence of the other returned paths is considered as the source of other traffic densities (in real life there can be other road users that are not using this system) and each of the routing results are therefore fed back to the system for updating the current density information.

A data structure Current Density is developed to store the information as well as to enable quick updates after finding and accepting each shortest path. Current Density is a list of sorted dictionaries. Edge IDs are used as the list index and the enter times are as the key of the dictionary. The traffic density of a road $r$ at time $t$ can therefore the denoted with $\text{CurrentDensity}[r][t]$.

Note that the default travelling speed will remain constant (see equation (8)) so only the current density will be updated. The idea of the update is to increment the traffic density of each involved edge in the returned path within the range of the corresponding time when the vehicle enters and leaves the edge. The update procedure is as follows:

**Algorithm 1** Current Density Update Algorithm

1: \textbf{let} $CD = \text{Current Density}, \ RP = \text{Returned Path}$
2: \textbf{for each} edgeID, $<\text{enter\_time}, \text{leave\_time}>$ in $RP$ \textbf{do}
3: \hspace{1cm} $\Delta = 1 / \text{Geographical Distance of edge with edgeID}$
4: \hspace{1cm} $CD[\text{edgeID}][\text{leave\_time}] = \text{Traffic density at time on or closest before leave\_time in CD}$
5: \hspace{1cm} $CD[\text{edgeID}][\text{enter\_time}] += \Delta$
6: \hspace{1cm} \textbf{for each} $time, traffic\_density$ in $CD[\text{edgeID}]$ where $\text{enter\_time} < time < \text{leave\_time}$ \textbf{do}
7: \hspace{2cm} $\text{traffic\_density} += \Delta$

In the final implementation, in order to reduce the size of Returned Path, the $\text{leave\_time}$ field is omitted and is obtained in run-time from the $\text{enter\_time}$ field of the next edge in Returned Path. For the final edge where there is no next edge to travel, the end time of the whole Returned Path will be used as the leave time. This strategy is feasible as long as the edges in Returned Path are sorted by the order of visited, i.e. by the enter time. In practice no extra sorting is needed as Returned Path is built with the visiting order of the A* traversal.

Although maximum time efficiency can be achieved by storing the densities for every second in the day in arrays and accessing the corresponding data by constant time direct array access, the amount of information stored will be enormous when such array is created for all edges. Taking C/C++ programming language as the implementation language, given that the current map of Hong Kong contains around 96500 edges, such caching method will create in a total of around $8 \text{ Byte} * 60 \text{ seconds} * 60 \text{ minutes} * 24 \text{ hours} * 96500 \text{ edges} = 6.67 \times 10^{10} \text{ Bytes} = 62 \text{ GB}$ of data in one day for all edges regardless of the number of requests handled.
On the other hand, the currently designed data structure gives each edge an empty ordered map during initialization and is only added with a key-value entry when there is a change in the traffic density of an edge, i.e. entering and leaving an edge contributes at least 1 change entry to the map. In other words, the amount of space used to store the traffic density information can be limited to $O(\text{number of edges in each path} \times \text{number of paths})$. For instance, in a day with total of 1000000 returned paths with an average of 50 edges in each path, the space required to store all the changes (assume they are in the worst case in which there are no overlapping changes and each change leads to an extra entry to reset the traffic density) is estimated to be around $(8\ \text{byte} + 8\ \text{byte} + 32\ \text{byte of entry overhead}) \times 2 \times 50\ \text{edges} \times 1000000\ \text{paths} = 4.47\text{GB}$, which is only around 0.07% of the array method and is far more manageable for ordinary servers.

Despite the gain in space efficiency, there are some compromises in the time efficiency. Almost all operations on Current Density now requires logarithmic time instead of constant time in direct access lists because of the tree nature of sorted dictionaries. All single updates in traffic density needs logarithmic time as well because of the traversal in the underlying binary tree structure of the ordered map and the rebalancing of trees when new entries are added. On the other hand, although binary search has to be used in an ordered dictionary for the retrieval of information given an enter time of the vehicle into the edge in the frequent cost calculation part of the routing algorithm, this actually enables quick search for the nearest traffic density to be used when there does not exist an entry of the enter time, while a more expensive linear search is needed for an unordered dictionary.

2.1.3 Calculation of Realized Travelling Time with Current Density

With Current Density in hand, the realized travelling time of an edge can be calculated with Equation (6). A binary search is first performed with the total realized travelling time on the edge in Current Density for the nearest density information. The found density is then fed into the equation to yield the realized travelling time. Notice that in very congested situations the density found can be larger than or equal to the maximum jam density when too much vehicles are allocated to this edge. In such scenarios the denominator in Equation (6) may become zero or negative, ultimately yielding undefined or negative travelling time. In order to prevent such adverse results, a guard for values found in Current Density is added, such that a large value indicating a congestion is returned whenever a density of value larger than 80% of the jam density is found.
2.2  Time-independent Incidents

2.2.1 Incidents and AffectedRoads

The time-independent incident information is stored and managed in two data structures named Incidents and AffectedRoads. They are used to handle the incidents as well as the relationship between the incidents and the affected edges respectively. Incidents is an unordered dictionary with an auto-incremented ID as the key and the predicted impact it will bring to the affected roads as the value. AffectedRoads is an unordered dictionary with the road ID (edge ID) as the key and a set of incident IDs as the value. Only edges with active incidents will have entries in AffectedRoads. Note that as the prediction of incident impact is not within the scope of this project, an external source like the incident reporter is relied upon to provide the incident location and its impact on the travelling time.

This combination of data structure will be used heavily in the cost calculation part during routing for providing incident modelling information in the function \( a(e) \). These include the presence of incident on the probed edge \( e \) and the impact of any present incident modelled as traffic density to denote how much the road will be jammed because of the incident. The value of \( a(e) \) can be given by the equation below:

\[
a(e) = \begin{cases} 
0, & \text{if } \text{AffectedRoads}[e] = \emptyset \\
\sum_{i} \text{Incidents}[i], & \text{otherwise, where } i = \text{IncidentIDs in AffectedRoads}[e]
\end{cases}
\]

Whenever there is an incident occurring on an edge \( e \), the value of \( a(e) \) will be the summation of the impact of all incidents happening on \( e \); if there are no incidents the value will be 0.

2.2.2 Reverse Geo-coding from Incident Location to Road ID

In the course of injecting incidents into the system, the incidents reporters should not have knowledge of the edge ID in the graph data used by the system. As a result, instead of reporting incidents by the affected edges directly, only the coordinates of the location of the incident will be reported to the system. Reverse Geo-coding is therefore needed to be performed by the system to identify the ID of the nearest (affected) roads from the location.

An intuitive approach is employed to serve this purpose: All edges in the graph are considered as a straight line, the corresponding linear equation and latitude and longitude ranges are found. The incident is then checked against each of edges with certain offset if it is in the “effective range” of the road. The approach of treating all edges is viable in the map data extracted for this project, although a road may contain turns and may not necessarily be straight. This is because the road turns in the OpenStreet Maps map data used in this project are geometrically modelled with multiple shorter straight edges. A scenario that the incident occurs on a turn such that only the very short edge is considered as affected may be yielded, but in such case the short edge is connected to the other short edges that form the turn – the travelling time of whole turn would have been increased by the affected short edge.
The approach of determining if a location is in the effective range of an edge is detailed as follows, this method is run on each edge in the graph while injecting an incident:

**Algorithm 2** Determine if a location is in the “effective range” of an edge

1. let \( p_1(x, y), p_2(x, y) = \) Coordinates of the start and end node of the edge, 
   \( q(x, y) = \) Coordinates of the target location 
   \( \text{offset} = \) Offset applied to the x and y axis to tolerate the location’s coordinates

2. if \( \min(p_1.x, p_2.x) - \text{offset} \leq q.x \leq \max(p_1.x, p_2.x) + \text{offset} \) 
   \( \min(p_1.y, p_2.y) - \text{offset} \leq q.y \leq \max(p_1.y, p_2.y) + \text{offset} \) then

3. if \( p_1.x == p_2.x \) \| \( p_1.y == p_2.y \) then return true

4. let \( m = (p_2.y - p_1.y) / (p_2.x - p_1.x), \epsilon = m \ast (-p_1.x) + p_1.y \)

5. if \( q.y - \text{offset} \leq m \ast q.x + \epsilon \leq q.y + \text{offset} \) then return true

6. if \( q.x - \text{offset} \leq (q.y - \epsilon) / m \leq q.x + \text{offset} \) then return true

7. return false

The offset variable enables the location to deviate from the straight line formed by the start and end of the edge by extending the line into a 2D shape illustrated as follows, let the original line be \( y = mx + c \):

![Figure 1: Illustration of Edge Effective Area](image)

As shown in Figure 1, the shaded effective area is bounded not only by the location of the start / end node but also the offset. This gives a geometrical model of a road in the real life and provides flexibility to the incidents to be identified as affecting the edges if they are not located precisely on the straight line between the start and end point of the edge. It can also effectively capture cases where an incident happens exactly at the junction of 2 roads or that when an incident happens very close to the start / end of the roads. As buffer space is reserved at the start / end node, the edges sharing the same affected junction would have overlapping effective area, thus allowing both edges to be identified as affected.
2.2 Backend Implementation

2.2.1 Technologies

The key technologies and libraries involved in the backend server are as follows:

2.2.1.1 C++ in CPPCMS Framework

The backend web server and algorithm modules are implemented with C++, a compiled programming language. As the source codes are compiled into low-level byte codes, programs written in C++ can be run efficiently by computer processors as there is no extra interpretation required. This is particularly useful in this project for offering short response time in the routing service.

To facilitate the development of the web server, a C++ web framework CPPCMS [7] is employed for the convenient framework and function libraries like URL mapping, request and response handling as well as JSON object manipulation. The C++ components can then be directly hooked into the web server without the extra needs of being wrapped as a plugin to be used and data conversion overhead in servers implemented with other programming languages like Python or JavaScript.

2.2.1.2 RoutingKit

RoutingKit [8] is an open-source third-party C++ library used in the server API implementations to provide convenient library functions for operations on map data. The included functions in this project include the Reverse Geocoding function for finding nearest nodes in the map graph for the origin and destination coordinates; as well as the file conversion function for extracting the map data like node coordinates and edge connectivity from OpenStreetMap files for the construction of the data structure used in the server.

2.2.1.3 Boost.Serialization

The Serialization module from the Boost C++ libraries [9] is used to serialize the constructed data structure. As constructing the data structure directly from an OpenStreetMap file takes relatively longer time, while deserializing a serialized file takes much shorter time, serializing the state of such data structure to a persistent file for later deserialization can save a lot of time during the initialization of the data to be used in the server. This can improve the efficiency of server development which requires frequent restart of the server.

The graph data will be residing in the memory of the server as a part of the web service during the entire runtime. This is to avoid slow retrieval of information from persistent storage and conversion overhead from the other data sources.
2.2.2 System Classes and Components

In this section, the high-level software architecture of the backend will be described. The details of the components will then be introduced.

MapGraph and IncidentManager are self-defined classes that encapsulate and stores all the information used in the system including the road map graph itself, the preprocessed distance table and the incidents injected. Router is another self-defined class that utilizes the information in MapGraph and IncidentManager to implement routing related functions like the heuristic function, realized cost function as well as the A* path finding function, which is the only function exposed to public usage. These classes should only have one instance in the whole application during run-time to avoid inconsistency of data.

IMSApp represents the web service layer implemented with the CPPCMS framework. It holds reference to the only instances of MapGraph, IncidentManager and Router throughout the runtime. The IMSApp is mainly responsible as the facet of the system to provide web APIs for any external client, which in the scope of this project is mainly the frontend application, to serve the O.D. requests as well as to receive requests of injection and removal of incidents. It relies on an external source to provide a serialized instance of MapGraph for it to boot and serve request with.

GraphBuilder is a separated command line applet developed as a user interface to trigger preprocessing of map data and serialization of MapGraph class to be used in the IMSApp server. This applet must be run at least once for the creation of the MapGraph required by IMSApp to run.
2.2.2.1 MapGraph Class

In order to store the graph information for path finding and information updating, a MapGraph class is designed. This data structure will reside on the system cache throughout the uptime of the backend application to support any operation on the graph. The basic class structure follows the SimpleOSMCarRoutingGraph class in the C++ RoutingKit library [8] with some additional fields. Table 1 shows the information and the corresponding data types that are stored in a MapGraph:

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>latitude</td>
<td>unsigned[]</td>
</tr>
<tr>
<td>longitude</td>
<td></td>
</tr>
<tr>
<td>head</td>
<td></td>
</tr>
<tr>
<td>first_out</td>
<td></td>
</tr>
<tr>
<td>geo_distance</td>
<td></td>
</tr>
<tr>
<td>default_travel_time</td>
<td></td>
</tr>
<tr>
<td>max_density</td>
<td>const double</td>
</tr>
<tr>
<td>current_density</td>
<td>map&lt;time_t, double&gt;[]</td>
</tr>
<tr>
<td>layers</td>
<td>layer_t</td>
</tr>
<tr>
<td>distance_tables</td>
<td>distance_table_t</td>
</tr>
</tbody>
</table>

Table 1: Data Stored in MapGraph

In such class, the node ID is used as the array index in latitude, longitude and first_out; while the edge ID information is used as array locations in head, geo_distance and default_travel_time.

The head and first_out array are used to record the edge connectivity information in the graph. The head array contains the destination of each edge; while the first_out array contains the first outward edge from each node. For instance, in a first_out with content \{3, 5\}, it is interpreted that node 0 has outward edges with ID 3 and 4. The origin and destination of edge 3 will then be 0 and head[3] respectively.

As discussed in the previous section, the maximum density is stored as a constant max_density. The Current Density structure is stored as an array of C++ STL (ordered) maps having a C++ time_t (i.e. number of seconds elapsed after 00:00 1/1/1970) as the key and a double float point number to represent the traffic density information as the value.
Layer_t and distance_table_t are two data types defined to store the preprocessed information for future use in speeding up the O.D. queries.

Layer_t is essentially a 2D vector of unsigned partition IDs. As the graph nodes are firstly divided into multiple layers and partitions before entering the preprocessing stage, each node / partition belongs to another partition except the “master” partition at the top. Layer_t records the parent of each node / partition in the partition hierarchy. Assume a node / partition with ID y on level x, y’s parent partition ID is can be accessed by layer_t[x][y]. By repeating such look-up, layer_t enables quick look-up for any ancestor of a node at any level in the next preprocessing stage as well as in the calculation of heuristics when handling queries.

Distance_table_t is a 2D vector of another self-defined data type entry_t. Entry_t represents an entry of the distance table computed during the preprocessing stage. It contains 3 fields, namely partition_distance, outbound_distance and inbound_distance. An entry is created for each member in each layer, including all partitions and nodes. This enables quick look-up for preprocessed distance information in the calculation of heuristics.
2.2.2.2 IncidentManager Class

The IncidentManager class is developed to encapsulate Incidents and AffectedEdges as well as to provide interfaces for the user of these data. The interfaces include injection and removal of incident, as well as the retrieval of the total impact on travelling time generated by all the incidents on an edge’s effective area. Table 2 shows the information and the corresponding data types that are stored in IncidentManager:

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_of_incidents</td>
<td>unsigned</td>
</tr>
<tr>
<td>incidents</td>
<td>unordered_map&lt;unsigned, unsigned&gt;</td>
</tr>
<tr>
<td>affected_roads</td>
<td>unordered_map&lt;unsigned, unordered_set&lt;unsigned&gt; &gt;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Return Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>inject_incidents(vector&lt;unsigned&gt; affected_edges, unsigned impact)</td>
<td>unsigned</td>
</tr>
<tr>
<td>remove_incident(unsigned incident_id)</td>
<td>unsigned</td>
</tr>
<tr>
<td>get_total_incident_impact(unsigned edge_id)</td>
<td>double</td>
</tr>
</tbody>
</table>

Table 2: Data and Methods in IncidentManager

Notice that the number of incidents is also encapsulated in IncidentManager as a log of total number of incidents that have ever been injected to the system. This number also serves as the unique ID of the newly injected incident. This is feasible because by nature this number never diminishes in value during the run-time of the system.

Incidents are stored in the Incidents structure which uses an unordered_map to speed up the retrieval of information with constant time access. The unsigned incident ID is used as the key and the unsigned impact on travelling time, which is essentially how much the travelling time will be lengthened by this incident”, is used as the value. AffectedRoads is also stored in an unordered_map of unordered_set. The unsigned edge ID is used as the key and the unordered_set stores the incident IDs of all the incidents that affects the edge.

For the return values of the exposed methods, inject_incidents returns the allocated incident ID to the incident reporter. This ID should be kept for the future removal of incident from the system. Remove_incident returns the number of incidents removed from the system, this can be used to indicate if the incident ID fed into the function does exist in the system and prevent unexpected deletion. Finally, get_total_incident_impact returns the total sum of impact contributed by all incidents affected the edge specified.
2.2.2.3 IMSApp

IMSApp is a C++ server developed with CPPCMS framework. It serves as a facet to provide different web API endpoints so that O.D. requests and incident management requests can be handled with the underlying model classes. In other words, IMSApp is the main component of this project for providing all the routing related functionalities to the users. All the input and output (if applicable) of the server is in JSON object format, because of the clarity in the data format and the fact that it is well-supported in the backend and frontend frameworks used in this project. The endpoints implemented are essentially the final delivered functions, as described in the following table:

<table>
<thead>
<tr>
<th>API Endpoint</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST /route</td>
<td><strong>Input:</strong> 2 pairs of longitude and latitude: origin and destination. <strong>Process:</strong> 1. Search for the corresponding closest nodes in the graph. 2. Perform path finding to find the quickest path. 3. Feed the found path back to the system to update Current Density. <strong>Output:</strong> The found path with nodes involved, enter time of each edge, start time and end time of the travel.</td>
</tr>
<tr>
<td>POST /reroute</td>
<td><strong>Input:</strong> 2 pairs of longitude and latitude: current location and destination. Original path returned from the server. <strong>Process:</strong> (This is a support functionality of routing, more details will be discussed in the later section.) 1. Search for the corresponding closest nodes in the graph. 2. Remove impact of original path from Current Density. 3. Perform path finding to find the quickest path. 4. Feed the found path back to the system to update Current Density. <strong>Output:</strong> The found path with nodes involved, enter time of each edge, start time and end time of the travel.</td>
</tr>
<tr>
<td>POST /incident</td>
<td><strong>Input:</strong> A pair of longitude and latitude: Incident location A positive number: Predicted impact of location on travelling time <strong>Process:</strong> 1. Find the affected edges of the incident location. 2. Create the incident entry in Incidents, get the incident ID. 3. Create/Add incident entries to AffectedRoads under the affected edges. <strong>Output:</strong> The incident ID.</td>
</tr>
<tr>
<td>DELETE /incident?incident={i}</td>
<td><strong>Input:</strong> i: The incident ID returned from the system. <strong>Process:</strong> 1. Remove the incident entry with ID i from Incidents. 2. Remove the incident entries in AffectedRoads, if an edge has 0 incidents left, remove the edge from AffectedRoads. <strong>Output:</strong> Nil, 200 - OK status code.</td>
</tr>
</tbody>
</table>

Table 3: API Endpoints available in IMSApp
2.2.2.3.1 Concurrency Problem

During implementation, it was found that CPPCMS was using a multithreaded approach to serve web requests. A thread pool is created with initialized worker threads each running an instance of IMSApp. Although all instances of IMSApp are referencing to the same instances of MapGraph and IncidentManager, this may still create a problem of inconsistency and starvation in the MapGraph and the IncidentManager as they are updated dynamically by multiple threads during the runtime of the server. In case of multiple concurrent requests coming into the server, multiple worker threads are supposed to be waken up by the framework to start processing the requests. In the design of the routing endpoint, update must be done after routing so that Current Density can reflect the latest usage of the roads and yield more useful routing results. As the scheduling policy of a multi-threaded environment is unpredictable, without extra mutexes or locks, it may happen in the worst scenario that part of the density update tasks from other threads take place during the routing phase and brings inconsistency to the data; or when all the threads are switched after the routing procedure, leading to the scenario that the updates are starved and are done only after all routing is finished. In this case, the routing process will be using the Current Density that is not updated at all and no time sensitivity will be shown in the results. The routing result will ultimately be unpredictable.

In order to avoid this problem, a static mutex lock is added to the IMSApp class and this lock must be obtained by the thread before entering the routing and updating procedures, or else the thread will be put to sleep waiting for the access. Note that the lock must be static so that all the IMSApp instances share the same lock. This ensures the atomicity of routing and updating such that every other request will always have the most updated Current Density and incident information to work with and time sensitivity can be shown. Although this method ensures the correctness and absolute consistency of the routing results, it causes severe lost in concurrency as now no two threads can simultaneously perform routing and updating, which means that all requests are served one by one. In the future, this may be improved by enforcing a conditional variable and a reader-writer lock, such that a certain amount of simultaneous routing can occur, and updates must be done atomically after this amount of routing. This can be feasible as one fed-back path may not cause too significant increase to the involved edges, temporary ignorance of them in Current Density may be a good enough approximation of the results, given that the updates will finally be performed.

An instance of IMSApp was deployed to the Ubuntu development server offered by the HKU CS Department for testing and experimental purpose for the end-to-end service with the frontend application. A deployment script was also written to automatically pull latest changes from the code repository and rebuild the application given that all dependencies are installed.
2.2.2.4 Graph Builder

Graph Builder is a C++ applet developed to perform precomputation of map data, and to create and serialize a MapGraph instance with the precomputed data for the IMSApp to use. As seen in Figure 3, the applet provides a simple user interface for users to perform generation and validation of MapGraph.

OpenStreet Maps’ Hong Kong map is chosen as the source of map data for it is free to use. A compressed binary version of the extracted OpenStreet Maps file with file extension of “.pbf” is required by the applet to extract the graph topology and the geographical information. The data in the “.pbf” file will first be extracted with the library function provided in RoutingKit, then partitioning and preprocessing functions will be called to compute the distance table as well as to create the MapGraph class. After preprocessing, the MapGraph class is serialized as a “.graph” file and is ready to be used or validated.

Simple validation can be done in Graph Builder by supplying the original .pbf file. The “.graph” file is first deserialized and the expected map information is extracted from the original file. The map information captured in the MapGraph will then be compared to the original information in the “.pbf” file. This validation step is particularly useful for checking “.graph” file that cannot be correctly deserialized correctly during the start-up of the IMSApp server because of corruption of data.
2.3 Function Design

There are 4 main functions in this application. While the end-to-end flow of each function will be covered in a groupmates report, this section will focus on the workflow of components in the backend.

2.3.1 Routing Service

This service is served on POST /route as shown in Table 3.

As shown in Figure 4, the workflow is unfolded as follows:

1. IMSApp receives the HTTP route request from an external source (e.g. the Frontend application) with the origin and destination location coordinates as parameters.
2. The mapGeoLocation function in MapGraph is called with the origin and destination coordinates as function parameters respectively. This performs Reverse Geocoding for the ID of the nearest node in the map graph data structure.
3. Routing is then performed by calling the route function of the Router with the origin node ID, destination node ID and the current time as the start time. The shortest path is returned.
4. The path is sent back to the client via HTTP as the request response.
5. Graph update is performed after the path is returned to ensure user experience. The injectImpactOfRoutedPath function of the MapGraph is called with the path to increment the density information.
2.3.2 Reroute Service

This service is served on POST /reroute as shown in Table 3.

Reroute Service is automatically invoked on the client side for periodically. The main purpose of this endpoint is to perform new O.D. requests for the user with the current location as the new origins.

This brings the benefit of rerouting user away from incidents that happen after the initial route is returned. In the initial settings, the system has no knowledge of returned paths and therefore can only prevent routing users into incidents when the incident appears before the routing request. This may not be ideal in a real-life use case when an incident occurs in the returned path after the path is returned, or that when many other vehicles suddenly arrive at a road segment earlier than this user. The user will not be notified and will ultimately experience a longer total travelling time in the traffic jam. The reroute functionality is therefore created for user clients to request for routing from the current position regularly and get an updated route calculated with the latest traffic information like the traffic density of the roads and the presence of incidents. Note that the newly calculated path will only be used and returned to the user when it costs less time than the remaining travelling time of the currently used path for maintaining the user experience.

In real-life, vehicle locations can deviate from what estimated in the initial route due to factors like road conditions and realized travelling speed. The routing algorithm depending on the Current Density cache may therefore be misled and returns non-optimal paths. The reroute function can therefore also act as a calibration of the density information stored in the system by resetting and updating the density impact brought by each vehicle with their latest position and the newly predicted locations. The Current Density cache can therefore reflect the realized traffic better and help the routing algorithm to return a path with better travelling time.
The detailed backend workflow of the reroute API endpoint is as follows:

1. IMSApp receives the HTTP request from the user with the current position, the destination and the old path as parameters.
2. IMSApp calls the mapGeoLocation function of MapGraph. This performs Reverse Geocoding for the current position and the destination to find the corresponding nearest node IDs in the map graph data structure.

As shown in Figure 5, the backend workflow of the reroute endpoint is unfolded as follows:

1. IMSApp receives the HTTP request from the user with the current position, the destination and the old path as parameters.
2. IMSApp calls the mapGeoLocation function of MapGraph. This performs Reverse Geocoding for the current position and the destination to find the corresponding nearest node IDs in the map graph data structure.
3. IMSApp calls the removeImpactOfRoutePath function of MapGraph with the old path as the parameter. The function clears the existing density impact that the currently used (old) path brings to the Current Density cache for the correct rerouting of the same user, or else the new impact created by recalculated routes will accumulate and yields incorrect results for other routing requests.

4. Ordinary routing is performed to find the shortest path between the current position and the destination by the Router.

5. The travelling time of the newly calculated path is then compared with the remaining travelling time of the old path. If the new path is faster than its old counterpart for more than N unit of time, where N is a self-defined constant, the new path will be adopted. If not, the old path will be adopted. The adopted path will then be returned the user client to finish the request.

6. IMSApp then calls injectImpactOfRoutePath function of MapGraph to perform updates in the Current Density cache with the adopted path.
2.3.3 Incident Injection

This service is served on POST /incident as shown in Table 3.

Figure 6 shows the flow of handling an incident report in the backend:

1. IMSApp receives the incident location and the predicted impact, then performs Reverse Geocoding to find the nearest edge and get the corresponding edge ID from MapGraph.
2. The incident is injected to the system with addIncident in IncidentManager. An incident ID is returned after creating all the entries.
3. APIController returns the incident ID to the incident reporter.

The incident reporter will have to keep the incident ID for the identification of the incident in the next stage of Incident Removal.
2.3.4 Incident Removal

Figure 7 shows the flow of handling an incident resolution report in the backend:

1. IMSApp receives the incident ID.
2. IMSApp calls removeIncident in IncidentManager and gets back the number of incidents removed.
3. IMSApp returns success to the incident reporter if the number of incidents removed is larger than 0, otherwise returns an error.
3 Results and Discussions

A variety of experiments were performed to evaluate the ability of the routing service in avoiding incidents and the roads that are congested by the other traffic. All experiments were performed on a bare metal Ubuntu 18.04 Bionic Beaver machine with Intel Core i5-8250U and 12GB of RAM. All the requests were generated and sent to the server locally with separated Python scripts for the easy aggregation and visualization of data. The server application was restarted for each set of experiments to ensure no previous traffic existed in the system and might potentially bring unexpected effect to the routing results.

3.1 Traffic Awareness

The travelling time of the path found being an indicator of the usefulness of the routing result, this section of experiments tries to compare the travelling time of routing with traffic awareness and that of routing without. The total travelling time for both cases are estimated as the sum of all the travelling time required for the involved edges, which can be calculated with Equation (6) as mentioned in previous sections. In the part of routing without traffic awareness, although the realized traversal cost calculated during routing is only obtained from the default travelling time, the impact of the path is still injected back to Current Density such that the estimated travelling time returned is still computed based on traffic information.

Three sets of experiments with different total number of requests (50, 500, 1000) were performed. Both routing with and without awareness were performed in all sets. Requests in the sets were sent in batch with very short pause between to simulate the real-life usage when multiple users request for the path of the same / similar O.D. pair. In order to test the ability of being aware of existing traffic and thereby avoiding routing into them, one single O.D. request was sent to the system repeatedly for multiple times, such that the traffic on the roads can be accumulated around the original path. The same O.D. longitude and latitude pair [114.17059, 22.29472], [114.15623, 22.33543] was used for all requests, i.e. both with and without traffic awareness experiments and in all number of requests.

For each set of experiments, apart from the travelling time returned for each request, the average travelling time for both cases of traffic awareness, the corresponding percentage improvement and the portion improved (percentage of requests that has lower travelling time then using no traffic awareness) are also of interest for evaluation. The same metrics were also considered for the upper 50% of requests to investigate the effectiveness of the method in more congested settings.

The experiment results of different sets are shown and discussed as the followings:
3.1.1 50 Requests Set

![Figure 8: Reported Travelling Time of Each Request in 50 Request Set](image)

<table>
<thead>
<tr>
<th></th>
<th>ATT (min)</th>
<th>ATT-U (min)</th>
<th>UP</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Awareness</td>
<td>6.72</td>
<td>7.24</td>
<td>9</td>
</tr>
<tr>
<td>Without Awareness</td>
<td>7.11</td>
<td>7.73</td>
<td>1</td>
</tr>
<tr>
<td>% Improved</td>
<td>5.11%</td>
<td>6.42%</td>
<td></td>
</tr>
</tbody>
</table>

*Table 4: Aggregated Results of 50 Request Set*

From Table 4, the average travelling time of routing with awareness is 6.72 minutes, which brings 5.11% improvement when compared to that without. For the upper 50% of requests, the average travelling time is expected to be higher than the that of all requests, because the same roads around the original path would have been used or jammed by other traffic in the lower 50%. In the case with awareness, the average travelling time is 7.24 minutes while that without is 7.73 minutes. A 6.42% of improvement can be seen in this case.

As shown in Figure 8, the curve of travelling times of every request for both case of awareness grows in a similar manner for such a small request set, except that the one with traffic awareness grown in a slower rate and peaked at around 7.7 minutes as compared to more than 8.5 minutes for that without. All paths returned with traffic awareness has travelling time lower than the expected travelling time of that without, thus 100% portion improved can be recorded for this set.

Notice the drops in travelling time of routes with traffic awareness in Figure 8. These drops exist because of clearance of traffic on the edges / roads used when the vehicles leave the road, so that the estimated travelling time on these roads drops and ultimately leads to a drop in total travelling time. This can only be achieved when traffic is spread to multiple roads (9 in this set, more in other sets) based on the traffic density so that each edge is occupied by fewer cars. The traffic density of these edges can then drop more easily because it only needs to digest fewer cars, yielding a shorter travelling time. On the other hand, without traffic awareness, all requests are routed to the same roads, leading to the circumstance that the traffic density of the edges will remain high until there is all vehicles finish their visit, creating a congestion and therefore
3.1.2 500 Requests Set

![Figure 9: Reported Travelling Time of Each Request in 500 Request Set](image)

<table>
<thead>
<tr>
<th></th>
<th>ATU</th>
<th>ATT-U</th>
<th>UP</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Awareness</td>
<td>11.8</td>
<td>14.7</td>
<td>81</td>
</tr>
<tr>
<td>Without Awareness</td>
<td>13.3</td>
<td>16.2</td>
<td>1</td>
</tr>
<tr>
<td>% Improved</td>
<td>10.8%</td>
<td>9.65%</td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Aggregated Results of 500 Request Set

* ATT = Average Travelling Time (minutes)
  ATT-U = Average Travelling Time for Upper 50% of requests
  UP = Number of Unique Paths returned

From Table 5, the average travelling time of routing with awareness is 11.8 minutes, which brings 10.8% improvement when compared to that without. For the upper 50% of requests, the average travelling time is 14.7 minutes while that without is 16.2 minutes. A 9.65% of improvement can be seen in this case, which is slightly lower than that of all 500 requests.

As shown in Figure 9, the travelling time of routing without traffic awareness is observed to be growing linearly with the number of request and peaks at around 18 minutes. Multiple spikes can be observed for the returned result of routing with traffic awareness, and these spikes yield results worse than routing without traffic awareness.

One major contributing factor of such spikes can be that all secondary paths chosen apart from the original path has at least one edge jammed at the time of the realized cost of the routing algorithm traversal, such that the jammed density defined in previous section is returned. In the initial settings all users are expected to start travelling immediately after receiving the path, and all requests are fired at the same time in the settings of the experiment environment. While this may not be the case in the reality, in real life these spikes may be reduced as the requests may not be as frequent. The spikes can be reduced by the user triggering re-route function in the application as well, as the actual driving speed of the users may not be the same as predicted in the system, the load on roads may be spread among the road more evenly instead of having the whole crowd travelling together on the same edges.
Despite the existence of these resolvable outlying spikes, it was found that this 500 requests set can achieve 89.4% portion improved with only 53 requests having travelling time worse than routing without travelling awareness, and all of them are the spikes mentioned above. If they are removed from the results set, the plot of reported travelling time for the remaining requests is as follows:

![Spikes-free Travelling Time for 500 Requests Set](image)

As observed from Figure 10, without the disturbance of the spikes, most the returned travelling time with traffic awareness grows much slower than the travelling time and has been covered in this plot. When compared with the linearly growing travelling time of routing without traffic awareness which peaks above 18 minutes, that with traffic awareness is almost growing like a log curve which peaks at around 12 minutes. It may be concluded that under most circumstances the travelling time yielded with traffic awareness is growing in a log-function-like manner.
3.1.3 1000 Requests Set

From Table 5, the average travelling time of routing with awareness is 14.8 minutes, which brings a significant 22.7% improvement when compared to that without. For the upper 50% of requests, the average travelling time is 17.8 minutes while that without is 24.7 minutes. An even more significant 28.2% of improvement can be seen in this case. This shows that routing with traffic awareness can achieve useful results even in larger scale request usage.

As shown in Figure 9, the travelling time of routing without traffic awareness is observed to be growing linearly with the number of requests. Like the 500 requests set, multiple spikes can be observed for the returned result of routing with traffic awareness, the number of spikes that are worse than routes without traffic awareness increases to 142, but this can be explained with the increased number of total requests. The portion improved in this set remains high - 85.8%. If the spikes are removed from the results set, the remaining 85.8% of travelling time behaves as follows:

From Figure 12, it is observed that the travelling time of routes with traffic awareness continues to grow following a log-like function and much slower than the linear growth of that without traffic awareness. Some spikes are still observed, they exist for the same reason explained above, but their value is still lower than their counterpart’s. Again, these spikes may be reduced with re-routing in the real-life use case.

Table 6: Aggregated Results of 1000 Request Set

<table>
<thead>
<tr>
<th></th>
<th>ATT</th>
<th>ATT-U</th>
<th>UP</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Awareness</td>
<td>14.8</td>
<td>17.8</td>
<td>220</td>
</tr>
<tr>
<td>Without Awareness</td>
<td>19.0</td>
<td>24.7</td>
<td>1</td>
</tr>
<tr>
<td>% Improved</td>
<td>22.7%</td>
<td>28.2%</td>
<td></td>
</tr>
</tbody>
</table>

* ATT = Average Travelling Time (minutes)  
  ATT-U = Average Travelling Time for Upper 50% of requests  
  UP = Number of Unique Paths returned
3.1.4 Summary

To summarize, routing with traffic awareness brings significant improvement in estimated travelling time when compared to routing without such awareness, especially in a larger-scale congested scenario.

By routing with traffic awareness, the cost of travelling on an edge calculated during routing must be larger than without traffic awareness when a vehicle is routed to this edge at the projected moment. By the optimization strategy of A*, edges with lower costs are therefore used with higher priority than the originally better edge, which is now relatively more congested, resulting in different final paths. The increased number of roads used to distribute the traffic (9, 81, 220 different paths used for the 3 sets) such that the traffic density on each edge can be diluted ultimately brings improvement to the travelling time.

It is noteworthy that such awareness can bring more improvement in congested scenarios of larger number of requests, as shown in the results for the 1000 requests set. This can be explained by log-function growth of travelling time by the number of request (vehicle) when compared with the linear growth of routing without traffic awareness. The improvement rate may not be as significant in smaller sample size (6.42% in upper half of the 50 requests set), but it became most significant in a congested situation of a larger sample size (28.2% in upper half of the 1000 requests set).
3.2 Incident Awareness

In this set of experiment, an O.D. request was first sent to obtain the original path and the estimated travelling time. After that the server was restarted, such that this original path can be cleared and will not have effect on future routing. An incident was then injected into the system on one of the roads involved in the original path. This O.D. request was then sent again for the alternative path and the new estimated travelling time. The above steps were repeated for the routing without incident awareness.

The origin and destination pair used in this experiment is from Hong Kong Cultural Centre to Cheung Sha Wan MTR Station Exit C2. The incident was injected on Lai Chi Kok Road next to Sham Shui Po Park with impact to travelling time of 3 minutes (180000 ms). The results are shown at follows:

The estimated travelling time of the path without incident awareness increased exactly by the value of impact brought by the incident to 8.62 minute from the original 5.62 minutes, this also shows that the incident is captured correctly. On the other hand, the path returned with incident awareness has estimated travelling time of 5.63 minutes, which is very close to the original travelling time. The incident awareness in this experiment saved 34.7% of travelling time. As seen in Figure 13, the alternative path with incident awareness is most identical with the original path and only tries to route away from the incident at almost the last turn before encountering the incident. This ensures that the travelling time can be minimized by sticking to a path that were the quickest without the incident as much as possible.
3.3 Combined Awareness

In the final implementation, both traffic and incident awareness are enabled so that estimated travelling time of the found path can be minimized by actively routing away from congested roads and incidents. This section shows how the two kinds of awareness work with each other and provide the user with the routing result.

The same O.D. pair the previous Incident Awareness section was used again, the first O.D. request was sent to the server to inject traffic and to get the path, then the second same O.D. request was sent to get the new path routed. After that the server was restarted to clear impact of the two paths. With these steps the original path and the path with traffic awareness can be obtained. The above procedure was then repeated with an incident injected to the system before sending the second request. The injected incident should be on an edge that is exclusive visited by the alternative path. A third path with both traffic awareness (aware of the original path) and incident awareness can then be obtained. Below are the results of such experiment:

Figure 14 shows that Path B follows mostly Path A for the first half to avoid the traffic injected in the original path, traffic awareness can be observed. In the second half of Path B, the path deviates from Path A and routes away from the incident before it reaches the destination. Notice that the final portion of Path B joins the original path, this was because the vehicle on the original path is estimated to have passed through the intersection edge at the time when the Path B first intersects with the original path, no congestion is presented so Path B is routed as so. This also shows a sign of successful traffic awareness.
4 Conclusions

This paper discussed the application part of the project, focusing on the design and implementation of traffic awareness and incident awareness which are included to provide useful routing results, as well as the details of the backend development. To conclude, this project has gained success in incorporating traffic and incident awareness into normal shortest path finding systems and providing improvements in routing results.

The traffic awareness feature of the system mainly relies on the equation derived from Greenshield’s Model to for the computation of travelling time of an edge based on its density at different time points. The Current Density data structure was also developed to allow efficient storage and access of density information of each edge at different time. The incident awareness feature was implemented with self-designed data structure Incidents and AffectedRoads to capture the incident impact of the edge travelling time as well as the corresponding roads that take the impact. A utility method of determining the affected edges of an incident based on its location was also developed by finding the effective area of each edge and comparing the location with every one of them. These features are essential to be combined with the preprocessed heuristics function to provide quick and useful routing service.

A backend server was implemented in C++ with CPPCMS web framework and libraries like RoutingKit and Boost.Serialization. It mainly uses 3 classes including the MapGraph class which holds all the graph related information, the IncidentManager class which holds all the incidents and the Router class which provides routing functionalities with data from the former two classes. With these 3 model classes, the web component provides 4 web APIs as the final project functionalities: Route, Reroute, Incident Injection and Incident Removal. The reroute function is specially designed for rerouting users in the real-life scenarios when incidents and congestions dynamically appear on a road that is involved in a returned path. It also serves as a calibration of Current Density data when users are not travelling as expected in the first place of routing. Locks were also introduced in the backend server to avoid consistency and starvation problem for Current Density updates in the multi-threaded server. An applet Graph Builder was also developed to perform preprocessing of OpenStreet Maps data and generate the MapGraph file needed to boot the server application.

3 sets of experiments were done to evaluate the improvement in predicted travelling time brought by the traffic awareness and the incident awareness. The first set was done to demonstrate the effect when the 2 kinds of awareness are available in routing. It was found that with traffic awareness, up to 22.7% time can be saved for 1000 requests when compared to that without. Also, for over 85% of the requests, the travelling time returned grows in a log function manner, which is much slower than the linear manner of routing without traffic awareness, which enables the system to yield significant improvement in larger-scale congested scenarios. The system was also found to be successfully routing away from injected incidents and finding paths that can give travelling time close to the original path while still having awareness of traffic. This shows satisfying effectiveness of the system in providing useful routing results to save users’ travelling time.
Currently, the system only concerns paths and traffic generated within the system itself and may not be able to perfectly model the actual traffic densities because of the lack of sensitivity of the road users that do not use this system. Besides, the Greenshields’ Model currently used to calculate travelling time from traffic density has limitations of undefined behavior in very congested situations. It also does not consider specifically the impact on traffic time on the other roads brought by the propagation of traffic congestions and vehicle queues, which may yield under-estimation of incident impact.

Moving forward, extra endpoints or methods may be added to the system to capture external traffic information so that the realized cost calculation function can reflect the actual travelling time more accurately. A hybrid model for different congestion severity [5] may also be employed in the calculation of travelling time so that the simple and efficient calculation in Greenshields can be preserved while possibly solving the undefined behavior in very congested situations. More effort may also be put on the incorporation of other map road factors like traffic lights, turns and one-way roads into routing such that more precise estimation of travelling time can be returned. It may also be worthy to investigate the attempts to obtain more accurate predictions of incident impact by considering the propagation effects of the congested vehicle queues [10].
5 References


