INTERIM REPORT

Incident Management System
For Intelligent City Development

20 January 2019

Group 2
Supervisor: Dr. Cheng, Reynold C.K.
Chow, Terence Yin Chak
3035271529
Kwong, Matthew Wang Shun
3035277030
Yuen, Calvin Hoi Man
3035244526
Abstract

Traffic congestion being one of the most serious problems faced by Hong Kong, this project aims to resolve this issue by providing a traffic-aware routing solution while taking live incident data into consideration. A new set of algorithms will be designed and proposed for traffic-aware and incident-aware pathfinding. In the end, a web service containing a backend server and client-side mobile application developed in programming languages and frameworks like C++ and React Native will be delivered as the final products of this project. Despite discovering some technical difficulties after some experiments, the project has gained some progress in the algorithm design, development of the backend server and the client-side mobile application. In the future, more effort has to be put into the algorithm proof and backend service implementation to ensure the project is in line with the planned schedule.
Acknowledgments

We would like to thank our supervisor, Dr. Reynold Cheng, for guiding us throughout the project.
Contents

Abstract 1
Acknowledgments 2
Contents 3
List of Figures 5
List of Tables 5

Introduction 6
  Background and Motivation 6
  Objective and Scope 7
  Deliverables 8
  Outline of Report 8

Methodology 9
  Algorithm 9
    Overview 9
    Related Works 10
    Layered Partitioning 12
    Preprocessing 14
    Query 18
    Current Travelling Time Calculation 21
    Traffic Density Information Update 22
      Current Density Affected By Returned Edges 22
      Time-independent Incidents 23
    MapGraph Data Structure 24
  Backend Service 26
    Technologies 26
      C++ in CPPCMS Framework 26
      RoutingKit 26
      Boost.Serialization 26
  Frontend Service 27
    Technologies 27
      React Native 27
      MobX 27
      Mapbox SDK 27
      Turf 27
  Function Design 28
List of Figures

Figure 1: Apply layered partitioning to a graph $G$ with $k = 3$ 13
Figure 2: The temporary vertices added and distances obtained during preprocessing 15
Figure 3: An example of $b^*(u, t)$ and $b(u, t)$ on the same graph where $U_i$ and $T_i$ reside within the same bound at level 1 19
Figure 4: Routing service system sequence diagram 28
Figure 5: Routing service backend sequence diagram 29
Figure 6: Routing service frontend sequence diagram 30
Figure 7: Reroute service system sequence diagram 33
Figure 8: Reroute service backend sequence diagram 34
Figure 9: Incident Appearance handling backend sequence diagram 36
Figure 10: Incident Resolution handling backend sequence diagram 37
Figure 11: Simulation system sequence diagram 38
Figure 12: HKSAR boundary (red line) 39
Figure 13: IMS home page 40
Figure 14: Enable location service confirmation dialog 40
Figure 15: Google Maps home page 40
Figure 16: Locations search result 41
Figure 17: Route Planning page 42
Figure 18: Google Maps Directions page 42
Figure 19: Google Maps “Add stop” function 42
Figure 20: Simulator drawer menu 43
Figure 21: Simulation type selection menu 43
Figure 22: Sample response from POST /route 44

List of Tables

Table 1: Data stored in MapGraph 24
Introduction

Background and Motivation

Traffic management is one of the most complex and crucial problems in developed cities like Hong Kong. Many online map services featuring route planning service are developed aim to solve this problem. With such a feature, it is now possible to quickly identify the shortest path between the desired starting point and destination. This enables drivers to spend a minimal amount of time and effort in finding the right direction while they concentrate on driving.

This problem 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 problem is popular and well studied. Throughout the researches, popular algorithms like Contraction Hierarchies[1] have been presented. These algorithms all work similarly. They first preprocess the graph without knowing the origin and destination. Then provide the shortest path with the given origin and destination. These algorithms assume the road network is static, so it only has to be preprocessed once and can be used for a large number of O.D. requests on the same road network.

However, the actual road network is not necessarily static, which there are 2 main scenarios causing 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.

Due to the existence of these scenarios, algorithms mentioned previously has lost their advantages, as their indexing strategies include initial edge weights (default travelling time) which are changed or lengthened in these case scenarios in their precomputed data and relies on them to speed up the search. That is, expensive recomputation has to be done whenever there is a change in the road network.

Road incidents and large-scale events being 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 [2]. 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 [3] (former ClearPath) provides the most similar service as this project is trying to deliver. Users first submit routing requests with origin and destination locations (O.D. requests) to the application server over the Internet, a path with the shortest travelling time will then be returned to the user device and be shown on a digital map. While calculating the path to be returned, TallyGo 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.

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 the time constraints, this project focuses on traffic management, more specifically, how to take real-time events into consideration while doing shortest path searches.

Understanding the limitations mentioned above, it also aims to deliver a web-based incident management service (IMS) that provides route suggestion functions focusing on the two main contributing factors of traffic jams – Large-scale events and Road Incidents, and to bridge the gap left by the other services:

Large-scale Events: Huge traffic flow and extra congestion may be generated from large-scale events and occasions like the Hong Kong Book Fair and large concerts held by famous singers. Therefore, the service aims to achieve routing which suggests efficient routes that minimize extra congestion created and ensure a reasonable travelling time for the users. This can be achieved by updating the map graph with the extra loads created to the roads by the served O.D. requests added to the edge heuristics.

Road Incidents: To avoid duplication of work, while the upstream incident detection mechanism relying on the traffic data provided by Hong Kong taxis will be handled by a peer group under the same supervisor Dr. Reynold Cheng, this project will put focus on using existing Hong Kong map data for the routing of O.D. requests assuming incident reports are available for access. The service tries to suggest a route which has minimal path segments passing through any congested areas affected by road incidents. Here, “road incidents” does not only cover car crashes, large events that generate huge traffic flow to nearby areas are also included.

Routing in Graph with Dynamic Edge Weights: This project tries to deliver a set of algorithms that brings sensibility of changes in edge weights in the road network brought by the other requests and the live incidents, and ultimately able to perform routing in a graph with dynamic edge weights.
Deliverables

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. Accepts parameters passed from the Backend service and returns the corresponding result back to the Backend in real 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 algorithm modules 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.

Outline of Report

The remainder of this report proceeds as follows. This report first describes the methodologies including the routing algorithm design, technologies involved and the workflow of different system functionalities. The current progress will then be discussed, followed by some proposed future actions.
Methodology

Algorithm

Overview

The algorithms used in the Incident Management System can be roughly divided into two parts, namely the routing algorithm and the update algorithm. The routing algorithm consists of a shortest path search on a road network map, which enables the system providing route planning service to the end users. The update algorithm on the other hand provides a mean of injecting both time dependent and independent modifications to the road network graph, which can be used to model previous routing influences and traffic congestions respectively.

There are quite a number of limitations and points of consideration when selecting and designing the routing algorithm used in the Incident Management System. From these criteria we propose the use of the A* algorithm[4] on a dynamic network with static lower bounds, and in order to provide speedups that achieve request resolving in real time, we further designed a way to obtain better lower bounds with the use of layered partitioning and precomputation on the road network graph. by combining the precomputed distances between partitions at different layers.

In order to reflect the dynamic properties of the road network, ways to perform graph update according to an accepted and returned path and according to time-independent incidents are also proposed. By the linear relationship between travelling speed and traffic density, the updates in edge weights (travelling time) caused by returned paths can be derived and represented as updates in traffic density and be calculated during runtime according to the intermediate total cost. A cache of current density is also proposed to store and update the traffic density information to be read during routing. For time-independent incidents, other data structures will be used to record their existence and the affected edges, so that they can be considered when calculating the edges costs.
Related Works

Shortest path search on graphs is a well-studied problem in computer science, where a huge number of researches and solutions have already been done. Since the classical algorithm proposed by Dijkstra in 1959 [5], the problem is considered to be solved in a relatively optimal manner. But still in some real-life applications such as road network path finding, Dijkstra’s algorithm along is still not fast enough to provide real-time searches. Thus, different speedup techniques have been developed upon Dijkstra’s algorithm throughout the years. In particular techniques such as Highway-Node Routing[6], Contraction Hierarchies[1], Multi-Level Dijkstra[7], Customizable Route Planning[8] and Precomputed Clustered Distances[9] provided huge speed ups to Dijkstra’s algorithm in a way that real time shortest path searches are possible.

Unfortunately for the development of the Incident Management System, a number of limitations do exist that prevent us from using the mentioned techniques directly in the system. One of the most significant limitations is that our system needs to support both the functionality of avoiding traffic incident on the road network, and reducing the impact from previous routed requests. This can be translated to be both static unpredictable increase in edge weights, and predictable but time dependent increase in edge weights in the shortest path problem. Most state-of-the-art speedup techniques mentioned above require either a completely static graph or at most minimal changes in terms of edge weights[1][8], or predictable changes at best[9] in order to function properly. This renders most of the speedup techniques not suitable for our situation. It is also important to note that most of the mentioned speedup techniques utilised precomputation to facilitate the speedups, where treating the graph with edges changes as a completely new graph requires redoing the precomputation, that in most cases, takes a long computation time.

The speedup technique we finally resorted in is the A* algorithm[4], which is a simple but also powerful modification to the original Dijkstra’s algorithm. Given the fact that the A* algorithm has been shown to work in a deterministic dynamic network[10], and unpredictable changes can be catered by simply triggering a reroute as the A* algorithm does not possess the need of any precomputation, it suits well with the limitations posed by the system.

In order to properly use the A* algorithm, a lower bound function must be given. The lower bound function provides a sense of direction to the algorithm when searching the shortest path, therefore given any lower bound function, as long as it satisfies a certain set of requirement[4], A* algorithm will always find an optimal solution, and more importantly, always expand fewer vertices than classical Dijkstra’s Algorithm. The accuracy of the lower bound function plays an important aspect of the speedup achieved by A* algorithm, as the accuracy determined the number of vertices A* expands, a hypothetically accurate lower bound can theoretically provide the most optimal runtime possible. In that sense, although using a simple lower bound function such as Euclidean distances would yield a certain amount of speedup, we would like to find a more advanced way such that it is closer to the optimal lower bound.
Our algorithm takes a similar approach of modeling the road network as other works mentioned above, the road network is treated as a directed graph, where each vertex represented a physical point location on the road network, in a way that the edge between a pair of vertices is the road segment spanning the distance of the two points. The edge weight represented the traveling time needed for a vehicle to completely travel the road segment. Although other matrices have been used by other works such as the number of traffic lights and the number of right / left turn needed[8], this definition should be sufficient for a general purpose path finding for typical usage.
Layered Partitioning

In order to obtain information for calculating the lower bounds, a layered approach is used in separating the original graph into partitions. The reason behind such a method is to proving detailed information about the path lengths at different layers of partitions, thus the information for lower bounds at each layer. For a directed graph \( G = (V, E) \), a total of \( l \) layers can be created. A layer consists of two kinds of information, firstly how the vertices of \( G \) is partitioned and secondly how the partitions relate with each other within the same layer. We denote a layer at level \( i \), \( i \leq l \) by \( L_i = (C_i, B_i) \), where \( C_i \) is the set of partitions at level \( i \), and \( B_i \) is the set of disjoint bounds for the partitions in \( C_i \) at level \( i \). The term bound here is used to denote a set of partitions, or equivalent, set of sets of vertices, restricting the search space during preprocessing of each layer.

The relationship between layers can be expressed as follows:

Consider the set of vertices \( V' \) contained within an arbitrary partition \( c_i \in C_i \) at level \( i \), we partition (denoted by a partition function \( p: \{v\} \rightarrow \{\{v\}\} \) \( V' \) into \( k \) sub-partitions \( c'_{i+1} \). A corresponding bound \( b'_{i+1} \) can then be defined as the set containing the sub-partitions \( c'_{i+1} \) built by the previously mentioned partition ci. \( C_{i+1} \) is the set of all sub-partitions \( c' \) built from each partition at level \( i \), and \( B_{i+1} \) is the set of all corresponding bounds \( b' \) of the said \( c' \). We would further define the base layer \( L_0 = (C_0, B_0) \), where \( C_0 = \{V\} \) and \( B_0 = \{C_0\} \), a layer with a single large set and bound that directly contains the whole original graph \( G \). This serves as the base case for the recursive definition of the layered partitioning process. The relations stated can be compactly summarized as the followings:

\[
\begin{align*}
C_0 &= \{V\} \\
C_{i+1} &= \bigcup_{c_i \in C_i} p(c_i) \\
B_0 &= \{C_0\} \\
B_{i+1} &= \{p(c_i) \mid c_i \in C_i\}
\end{align*}
\]

The full partitioning process terminates at level \( l_{max} \) where all partitions \( c_i \in C_i \) are in their smallest possible form, such that \( c_i \) contains only one vertex \( v \in V \), and can no longer be further partitioned into smaller sub-partitions. It can be easily shown that although further partitioning is possible after level \( l_{max} \), it provides little to none new useful information in regards to calculating the lower bounds. Furthermore, although it is beneficial to achieve partitions up until level \( l_{max} \), in practical cases we should select a parameter \( l \leq l_{max} \) such that the precomputed data generated during preprocessing would take up a reasonable size of storage spaces. Fig. 1 illustrates an example of the layered partitioning process and the resultant layers from the process.
The algorithm of the partitioning a graph \( G = (V, E) \) into layered partitions with \( l \) levels can be presented as follows:

**Algorithm 1** Layered partitioning algorithm

1. \( C_0 = V \)
2. \( B_0 = \{ C_0 \} \)
3. for \( i = 1 \) to \( l \) do
   4. \( C_i = \emptyset \)
   5. \( B_i = \emptyset \)
   6. for each \( c_{i-1} \in C_{i-1} \) do
      7. let \( p(c_{i-1}) \) be the set of sub-partition created from \( c_{i-1} \)
      8. \( C_i = C_i \cup p(c_{i-1}) \)
      9. \( B_i = B_i \cup \{ p(c_{i-1}) \} \)
Preprocessing

Preprocessing is the first stage of the routing algorithm, where information of the original graph was processed and stored that would support the acceleration of the query. Given a graph $G$ where layered partitioning has been done, we would want to achieve lookups that give the shortest possible travelling time from partition $m_i$ to partition $n_j$ at level $i$, the shortest possible travelling time for $m_i$ to exit the bound that contains itself, as well as the shortest possible travelling time for reaching $m_i$ from beyond the bound containing $m_i$. The lookup operations for these values should take approximately constant time to complete. This can be best done by compute / distance tables, one table for each layer catering the partitions at that layer.

For the computation of each of the distance tables, we borrowed the same method used by the Precomputed Cluster Distances[9] along with some slight modifications, which can be described as follows:

For each partition $m_i$ of layer $L_x$, a temporary vertex $v_w$ was added in the original graph $G$ that is connected to all border vertices of $m_i$ with zero weight edges. Another temporary vertex $v_M$ was also added in the original graph $G$ that is connected to all border vertices of the corresponding partition $m_{x+1}$ at layer $x+1$ of bound $M_x$, where $M_i$ is the bound containing $m_i$. A total of three single source shortest path search, such as Dijkstra’s Algorithm, is taken place after the temporary vertices have been set up. These three searches are used to find the following information:

1. Minimal distance from a partition to another in the same layer within the same bound
2. Minimal distance from a partition to the border of its bound
3. Minimal distance to a partition from the border of its bound

The first single source shortest path search originated from $v_w$ is performed on graph $G$. The temporary vertex $v_M$ should be ignored in this search. The search can then be prematurely stopped when all partitions $n_j \neq m_i$ that was contained in the same bound as $m_i$ was reached, or more precisely, one of the vertex from each partition $n_j$ are reached. The second single source shortest path search is similar to the first, but instead of targeting all partitions $n_j$, the search targets only the vertex $v_M$ instead. The search can be prematurely stopped when $v_M$ is reached. The third single source shortest path search originates from $v_M$ on graph $G$ that target $v_w$. Same as the previous search, this search can also be prematurely stopped when $v_w$ is reached. The shortest path distances from $v_w$ to all $n_j$, from $v_w$ to $v_M$ and from $v_M$ to $v_w$ are recorded as entries for $d(m, n)$, $d_{in}(m)$ and $d_{out}(m)$ in distance table for layer $L_x$. The following special cases are also added to the distance table:

1. $d(m, n) = 0$ if $m = n$
2. $d(m, n) = \text{undefined}$ if $M \neq N$, where $M, N$ are the bound containing $m, n$

Fig. 2 illustrates the relationship of the temporary vertices and the distances obtained with the partition involved.
This process is repeated for each partition at each layer, that would finally give a distance table for each layer that contains precomputed information for each pair of partitions within the same bound. For the purpose of our algorithm, it is essential to consider the graph $G$ with the edge weight as their least possible traveling time when doing the single source shortest path search. This is done to ensure the precomputed data qualifies being a heuristic for the searching algorithm used in the query.

For a layered partition graph with $l$ layers and each partition processes separate a partition into $k$ sub-partitions, The precomputed data from the preprocessing stage can be shown to take $\Theta(k^l)$ storage space, and can be further shown to take up $e\left(\frac{1-k^{l+3}}{1-k} - k - 1\right)$ space, where $e$ is the space needed for storing each entry in the distance table.

**Lemma 1.** The precomputed data takes $e\left(\frac{1-k^{l+3}}{1-k} - k - 1\right)$ space.

This proof is simple but we decided to include it nevertheless, as this may serve as an entry point for implementing the storage mechanism for the precomputed data.

**Proof.** Trivial data such as those evaluated to 0 or undefined can be omitted, such that those distances can be excluded from the distance table and instead retrieved by a simple if statement. Thus excluding them, the only data required for storage are those actually contain distance information. From the definition of the layers, partitions and bounds, we know for a fact that level 0 should not contain non-trivial information. Level 1 contains $k$ partitions, thus requires a total of $k^2$ number of entries. In the next level we create $k$ sub-partitions for the $k$ bounds, thus requires a total of $k^3$ number of entries. Then the total number of entries would be a geometric sequence of
\[\begin{align*}
\sum_{i=0}^{l+2} k^i &= \sum_{i=0}^{l+2} k^i = \sum_{i=0}^{l+2} k^i - k - 1 \\
&= \sum_{i=0}^{l+2} k^i - k - 1 \\
&= \frac{1 - k^{l+3}}{1 - k} - k - 1
\end{align*}\]

The impact on the effectiveness of the searching algorithm by the selection of parameters \(l\) and \(k\) will be further discussed in a later part when we experiment in the algorithm.

The following presentation of the preprocess algorithm contains the use of a normal single source shortest path search and a slightly modified Dijkstra’s Algorithm that suits the purpose of our algorithm. For a directed graph \(G = (V, E)\) with weight function \(w\), assuming layered partitioning is done on the graph such that \(L_i = (C_i, B_i), i \leq l\) is known. From that the algorithm for the preprocessing stage can be presented as follows:

**Algorithm 2 Preprocess algorithm**

1. for \(i = 1\) to \(l\) do
2. initialize distance table \(DT_i\)
3. for each \(m_i \in C_i\) do
4. let \(m_i'\) be the set of border vertices of \(m_i\)
5. \(N_i = C_i \setminus \{m_i\}\)
6. let \(v_m\) be a new temporary vertex
7. let \(e_w\) be a set of zero weight edges from \(v_m\) to each vertex of \(m_i'\)
8. \(V = V \cup \{v_m\}\)
9. \(E = E \cup e_w\)
10. for each \(v \in V\) do
11. \(\text{dist}[v] = \infty\)
12. \(\text{prev}[v] = \text{Nil}\)
13. for each \(n_i \in N_i\) do
14. \(\text{rep}[n_i] = \text{Nil}\)
15. \(S = \emptyset\)
16. \(Q = V\)
17. while \(Q \neq \emptyset\) do
18. let \(u \in Q\) be a vertex with the minimum \(\text{dist}[u]\)
19. \(Q = Q \setminus \{u\}\)
20. let \(c \in C_i\) be the partition containing \(u\)
21. if \(c \in N_i\) and \(\text{rep}[c] \neq \text{Nil}\)
22. \(\text{rep}[c] = u\)
23. if \(\forall n_i \in N_i, \text{rep}[n_i] \neq \text{Nil}\)
24. break
25. for each \(v \in \text{neighbours of } u\) do
26. if \(\text{dist}[v] > \text{dist}[u] + w(u, v)\)
Ideally, if the road network itself does not encounter any changes, such as introducing or removing road segments, preprocessing should only be done once during the course of deployment. The information acquired from preprocessing will be used later to facilitate the search speedups.
Query

Query is the part where most of the time the Incident Management System would spend on. In contrast to both layered partitioning and preprocessing where both occur at the start of deployment and major maintenance, query is a lifelong cycle of awaits and responses during the course of deployment. The main functionality the system would provide in this stage is the resolution of origin to destination requests from the end users.

As mentioned above, we would be using the A* algorithm for searching the shortest path for each origin to destination request. The key working principle for A* algorithm can be expressed as an evaluation function that dictates the new vertex being relaxed by the algorithm. We rewrite the evaluation function from [4] to the form of

\[ f(s, u, t) = g(s, u) + h(u, t) \]

where given the start vertex \( s \), target vertex \( t \) and the evaluating vertex \( u \), \( g(s, u) \) is the current cost function found at the moment of evaluation from \( s \) to \( u \), and \( b(u, t) \) is the estimated lower bound from \( u \) to \( t \). From here, we would like to provide our own definition for both the lower bound function and the current cost function that matches the needs for the system. Both functions assumed that the original graph has already been properly partitioned and preprocessed beforehand, such that all information needed would be available to be used.

The lower bound function \( b(u, t) \) we have developed can be expressed as the followings:

Let \( U_i \) and \( T_i \) be the partition that contains \( u \) and \( t \) at level \( i \) respectively, the lower bound from \( u \) to \( t \) can be calculated by the sum of for all the minimal distance between \( U_i \) and \( T_i \) if they reside within the same bound, and both the minimal distances of from \( U_i \) to the border of its bound and to \( T_i \) from the border to its bound if they do not reside within the same bound. For simplicity the lower bound function can be summarized as follows:

\[
  h(u, t) = \sum_{i=0}^{l} d_i, \text{where } d_i = \begin{cases} 
    d(U_i, T_i) & \text{if } \exists b_i \in B_i, \{U_i, T_i\} \subseteq b_i \\
    d_{\text{out}}(U_i) + d_{\text{in}}(T_i) & \text{otherwise}
  \end{cases}
\]

where \( d(U_i, T_i) \), \( d_{\text{out}}(U_i) \), \( d_{\text{in}}(T_i) \) are retrieval functions that read precomputed data stored in the distance tables obtained from preprocessing. It can be shown that with this definition, the admissibility requirement[4] for a lower bound function is satisfied, thus ensures the algorithm can arrive at the optimal solution reliably.
Lemma 2. \( b(u, t) \leq b^*(u, t) \)

The aim of this proof is to show that the lower bound \( b(u, t) \) we suggested do not overestimate the actual cost \( b^*(u, t) \) for any pair of vertices, which is the sole definition for the admissibility requirement poses by the A* algorithm.

**Proof.** Consider the actual shortest path \((u, \ldots, t, v)\) from any vertex \( u \) to any vertex \( t \), and \( U_i \) and \( T_i \) be the partition that contains \( u \) and \( t \) at level \( i \) respectively, and level \( m \leq l \) being the first level that \( U_m \) and \( T_m \) are contained within the same bound. We are able to split the path into multiple sub-paths \((u, \ldots, u'_i), \ldots, (u'_{i+1}, \ldots, u'_l), (u'_{i+1}, \ldots, t'), \ldots, (t', \ldots, t), \ldots, (t', \ldots, t), \ldots, (t', \ldots, t), \ldots, \) where \( u_i \) denotes the first vertex the path reach that belongs to a border vertex of \( U_j \), and \( t_i \) denotes the last vertex the path reach that belongs to a border vertex of \( T_j \). The possibility of such split of the path can be easily justified by the fact that in order for a path leaving or entering a partition, it must pass through the border vertices of the partition at least once, thus for any path of \( u \) reaching \( t \), the path must pass through all partitions from level \( l \) to \( m \), and in turn their border vertices at least once. Therefore, \( b^*(u, t) \) can be split into multiple parts corresponding to each sub-path that \( b^*(u, t) = b^*(u, u') + b^*(u', t') + \ldots + b^*(u', t') \). Expanding \( b(u, t) \) by the definition yields the expression \( b(u, t) \leq \sum d_{out}(u) + \ldots + d_{out}(u) + d(U_m, T_m) + d(U_m, T_m) + \ldots + d_{out}(T) \).

**Figure 3:** An example of \( h^*(u, t) \) and \( h(u, t) \) on the same graph where \( U_i \) and \( T_i \) reside within the same bound at level 1.

Comparing \( b(u, t) \) and \( b^*(u, t) \) can be done term by term

\[
\begin{align*}
h(u, t) &= 0 + d_{out}(U_i) + \cdots + d_{out}(U_m) + d(U_m, T_m) + d_{in}(T_m) + \cdots + d_{in}(T) + 0 \\
h^*(u, t) &= h^*(u, u'_i) + h^*(u'_i, t'_i) + \cdots + h^*(u'_i, t'_i) + h^*(t'_i, t'_i) + \cdots + h^*(t'_i, t'_i) + h^*(t'_i, t)_i + h^*(t)_i
\end{align*}
\]

that either the comparisons are trivial, or must be true since \( d(v, v), d_{out}(v) \) and \( d_{out}(v) \) are guaranteed to be having the least cost possible through preprocessing. Furthermore, for any level \( m' < m \), \( d(U_m, T_m) \) must evaluate to 0 and do not contribute to \( b(u, t) \) at all. Therefore, \( b(u, t) \leq b^*(u, t) \) must hold. □

With this definition of the lower bound function, two effects can be observed that how the function affect the priority for relaxing vertices. This provides us a slight insight that how we should partition the vertices within each bound during partitioning. The first effect the lower bound function provides is that it would prioritize vertices outwards toward the border of the partition if the target partition cannot be
reached within the bound in the current layer. The second effect is that if the target partition can be reached, a clear direction can be then provided and thus prioritize the vertices along the target direction. Since precomputed information is limited with bound, it can be concluded that for this lower bound function to work more efficiently, the partition criteria should be focused on dividing the vertices into many partitions, where the number of partitions containing border vertices of the bound should be small. The selection of the partition algorithm would be discussed in the future.

The current cost function $g(s, u)$ for calculating total travelling time we have decided to use can be described as follows:

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

where $u'$ is the previous vertex of $u$ along the expanding path from $s$ to $u$. This is basically a slight modification of the current cost function from the classical A* algorithm that supports time dependent edge weights. We distinguished the original edge weight retrieval function $w(e)$, which retrieves the minimum possible (legal) travelling time, from a time dependent edge weight retrieval function $w'(e, t)$ which aims at calculating the edge weight of edge $e$ at a specific time $t$. The new weight retrieval function can be expressed as:

$$
w'(e, t) = \frac{w(e)}{1 - \frac{\rho(e, t)}{\rho_0(e, t)}} + a(e)
$$

Within the formula encapsulates two different types of graph modifications the road network can encounter. Firstly there are the influences induced by previous routes, is computed by the time dependent density retrieval function $\rho(e, t)$, the time dependent maximum density retrieval function $\rho_0(e, t)$, and the original weight retrieval function $w(\cdot)$. The derivation will be discussed later in the Current Travelling Time Calculation section. Secondly, there are the impacts done by a traffic incident, and is simply represented by a time independent modifier function $a(\cdot)$. The details again will be discussed later in the Time-independent Incidents part.

It is important to note that, in order for the A* algorithm to work properly, the lower bound should still be admissible. This must also remain to be the case even if we use $w'(e, t)$ instead of $w(\cdot)$ to add time sensitivity to the current cost function, such that $w'(e, t) \geq w(\cdot)$ and therefore the lower bound remains an underestimate of the actual cost. The easiest way to do this is to ensure the following properties:

1. $a(\cdot) \geq 0$
2. $\rho_0(e, t) \geq \rho(e, t) \geq 0$, such that the 1st term has value at least $w(\cdot)$. 
Lemma 3. \( a(e) \geq 0 \)
Consider the following 2 cases of incident presence, refer to Time-independent Incidents section for the equation of \( a(e) \):
1. There is no incident present on the edge \( e \): \( a(e) = 0 \).
2. At least 1 incident occurs on \( e \). Since the impact is modelled as increase in traffic density, the individual impact value must be \( \geq 0 \), so as the summation of them.
Therefore \( a(e) \geq 0 \). \( \square \)

Lemma 4. \( \rho_0(e, t) \geq \rho(e, t) \geq 0 \), such that the 1st term has value at least \( w(e) \).
As the \( \rho(e, t) \) denotes density values and is only incremented but never decremented in graph updates, the value must be \( \geq 0 \). As shown later, \( \rho_0(e, t) \) is a constant with value \( > 0 \), the minimum value of \( \rho(e, t) / \rho_0(e, t) \) therefore is 0 and the maximum value of the denominator will be 1. On the other hand, as \( \rho(e, t) \) is capped at \( \rho_0(e, t) \), the maximum value of \( \rho(e, t) / \rho_0(e, t) \) is at most 1 and the minimum value of the denominator is 0. In such case, the edge is jammed and \( w'(e, t) \) is set to return \( \infty \). Therefore, the range of the 1st term becomes \( [w(e), \infty) \). \( \square \)

With the properties met, \( w'(e, t) \geq w(e) \) will be established and thus the real future cost \( b^*(u, t) \geq b^*(u, t) \geq b(u, t) \), maintaining the admissibility of the lower bound function. \( \square \)

Current Travelling Time Calculation
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 [11]. This model is widely used in the macroscopic traffic analytics field for it being simple yet fairly accurate. The model suggests the following relationship, where \( v = \) speed, \( v_{\text{max}} = \) max speed, \( d = \) traffic density and \( d_{\text{max}} = \) max traffic density:

\[
v = -\frac{v_{\text{max}}}{d_{\text{max}}} d + v_{\text{max}}
\]

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 relationship between travelling time and traffic density can be found. In order to achieve this, the equation can be further arranged as follows, where \( t = \) travelling time, \( t_{\text{max}} = \) default travelling time and \( g = \) geographical length of the road:
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. The value of default travelling time remains can easily be calculated as shown above with the legal maximum speed. For the maximum traffic density, as there is not enough data available to find such information for each road in Hong Kong, such value is estimated by the followings with the normal sized vehicle length $l$ of 5m (derived from the standard parking space governed by the Planning Department [12]). Let $n_{max}$ = 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.

$$d_{max} = \frac{n_{max}}{g}$$

$$d_{max} = \frac{\frac{g}{l}}{g}$$

$$d_{max} = \frac{1}{l}$$

As shown above, the maximum traffic density for all roads can be deduced as a constant value of $\frac{g}{l}$ regardless of the geographical length. Therefore, the only variable that current travelling time depends on becomes the current traffic density of the road, i.e. how many other vehicles are travelling on this road currently.

Traffic Density Information Update

Current Density Affected By Returned Edges

The current density of each edge at different time points must be recorded to provide time sensitivity to the routing, such that the algorithm has knowledge of the existence of the other returned paths and ultimately prevent the routing algorithm from routing to overly popular roads that are used by many other paths. Therefore, a data structure Current Density is needed to store the information as well as to do updates after finding and accepting each shortest path. Note that the default travelling speed will remain constant and only the current density will be updated as the current travelling time is calculated.
during the runtime (refer to Query section). 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 entry and leave time. The update procedure is as follows:

**Algorithm 3 Current Density Update Algorithm**

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

**Time-independent Incidents**

Time-independent incident information is stored in 2 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 hashmap with an auto-incremented ID as the key and the predicted impact it will bring the affected roads as the value. AffectedRoads is an unordered hashmap with the road ID (edge ID) as the key and a set of incident IDs as the value. Only edges with active incident 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 such information.

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:

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

Whenever there is an incident occurring on the road, 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.
MapGraph Data Structure

In order to store the graph information for path finding and information updating, a MapGraph data structure is designed. This data structure will reside on the system cache throughout the uptime of the backend application to support any operations on the graph. The basic class structure follows the SimpleOSMCarRoutingGraph class in the C++ RoutingKit library 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></td>
</tr>
<tr>
<td>longitude</td>
<td></td>
</tr>
<tr>
<td>head</td>
<td>unsigned[]</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>double</td>
</tr>
<tr>
<td>current_density</td>
<td>map&lt;time_t, double&gt;[]</td>
</tr>
<tr>
<td>incidents</td>
<td>unordered_map&lt;unsigned, double&gt;</td>
</tr>
<tr>
<td>affected_rows</td>
<td>unordered_map&lt;unsigned, set&lt;unsigned&gt; &gt;</td>
</tr>
</tbody>
</table>

Table 1: Data stored in MapGraph

In such data structure, the node ID information is used as array locations 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 Edge Update section, the maximum density is stored as a constant max_density, while the current density information is stored in current_density as an array of ordered map with the key (critical time point) as time_t and value (density) as a double. The current array of ordered map structure is chosen because of the followings:
Incident Management System - Interim Report

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. Given that the current map of Hong Kong contains around 96500 edges, such caching method will create in a total of around $8 \times 60 \times 60 \times 24 \times 96500 = 6.67 \times 10^{10}$ Bytes = 62GB of data in one day for all edges regardless of the number of requests handled.

On the other hand, the currently designed data structure current_density 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 only the $O(number\ of\ edges\ in\ each\ path \times\ number\ of\ paths)$. For instance, in a day with total of 10000 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 $(8\ \text{byte} + 8\ \text{byte} + 32\ \text{byte of entry overhead}) \times 2 \times 50\ \text{edges} \times 10000\ \text{paths} = 45.8\text{MB}$, which is only around 0.0007% of the array method and is far more manageable for ordinary servers. Despite the gain in space efficiency, there is some compromise in the time efficiency. Almost all operations on current_density now require logarithmic time. Binary search is needed for the earliest traffic density information available given an enter time of the vehicle into the edge in the frequent cost calculation part of the routing algorithm. All the updates in traffic density also need logarithmic time as well because of the traversal in the underlying binary tree structure of the ordered map.
Backend Service

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

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 is employed for the convenient framework and function libraries like URL mapping, request and response handling as well as JSON object manipulation. The C++ algorithm modules can then be directly hooked into the web server without the extra need of being wrapped as a plugin to be used in servers implemented with other programming languages like Python or JavaScript.

RoutingKit
RoutingKit is an open-source third-party C++ library used in the server API endpoint implementations to provide convenient library functions for the 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.

Boost.Serialization
The Serialization module from the C++ Boost library is also 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.
Frontend Service

Technologies
There are several key technologies and libraries involved in this mobile application.

React Native
React Native is a cross-platform mobile application framework developed by Facebook, which allows supporting both Android and iOS platforms in a tight schedule. React Native has relatively higher performance than similar frameworks and has gained its popularity rapidly in recent years.

MobX
MobX is a state management library for JavaScript, which also supports React Native. The main reason for using a third party state management library is because the default state management mechanism of React Native has unsatisfied performance, while MobX provides a great performance improvement.

Comparing to Redux, which is generally a more popular choice. MobX is more suitable for small to medium size project like this project due to its simplicity and ease of use.

Mapbox SDK
Mapbox is a provider of custom online maps for websites and applications. It provides a development toolkit which allows a customizable map view to be embedded into a mobile application. The decision of choosing Mapbox SDK instead of the more popular Google Maps API is, in fact, a compromise. The terms of service of Google Maps [13] has explicitly forbidden the creation of “a substitute of the Google Maps Core Services”, which includes route planning.

However, despite being less popular, Mapbox SDK is more customizable than Google Maps API in terms of visualization. The only limitation is the scarcity of documentation which causes a few setbacks during the implementation process.

Turf
Turf, or Turf.js is a JavaScript library for spatial analysis. It provides spatial operations and helper functions for creating GeoJSON data object. GeoJSON is a special form of JSON object which is used to represent map objects like paths, polygons, and boundary boxes in Mapbox SDK.
Function Design

There are 3 main functions in this application. The remaining sections of this part present the general workflow of each function.

Routing Service

Routing service is the most common function users will be using. This single endpoint first finds the shortest path, then it updates the graph according to the path found. As shown in Figure 4, the normal workflow (a.k.a main success scenario) is as follow:

1. User inputs an origin and a destination
2. Frontend service sends a HTTP request to the backend service with the input origin and destination
3. Backend service makes a function call to the implementation of the algorithm with the input origin and destination as arguments
4. The implementation of the algorithm returns a path
5. The implementation of the algorithm goes through update stage
6. The implementation of the algorithm returns the path to the backend service
7. Backend service returns the path to the frontend service
8. Frontend service stores the path for the reroute service (see Reroute Service)
9. Frontend service presents the path to the user
Backend

The detailed flow of the route function between the backend and the algorithm modules is presented in this section.

As shown in Figure 5, the workflow is unfolded as follows:

1. **APIController** receives the HTTP route request from an external source (e.g. the Frontend application) with the origin and destination location coordinates as parameters.

2. **APIController** calls the `mapGeoLocation` function of the RoutingKit library 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. **APIController** then performs routing by calling the `odRequest` function of the RoutingModule with the origin node ID and destination node ID. The shortest path is returned once it is found.

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 update function of the UpdateModule is called with the path to increment the density information.
Frontend

This section presents a more detail workflow of routing service in frontend.

As shown in Figure 6, the normal workflow is as follow:
1. User presses the direction button in the application
2. The application presents the direction page
3. User enters a keyword to search for a location
4. The application makes a HTTP request to Mapbox Geocoding service with the input keyword
5. Mapbox Geocoding service returns a list of locations matching the keyword
6. User picks one of the locations
7. The application adds the selected location (a stop, waypoint) to a list

*System repeats step 3 - 7 until there are 2 stops in the list (i.e. 1 origin and 1 destination)*
8. The application sends a HTTP request to the backend service with the input list of stops
9. Backend service returns a path (in the form of a list of edges)
10. The application converts the path into a list of vertices
11. The application makes a HTTP request to Mapbox Map Matching Service with the list of vertices
12. Mapbox Geocoding service returns a new list of vertices
13. The application calls `lineString` helper function from Turf to convert the list of vertices to polyline string
14. The application draws the polyline on the map view and presents it to the user

Notes
Step 3 - 7: The design of the frontend service in fact supports more than 2 stops (i.e. multiple waypoints between the origin and destination). However, due to the time constraints, this feature is eventually not implemented.

Step 4, 5: Mapbox Geocoding service provides a forward geocoding service. It takes a string as keyword and a number of searching criteria, and return a list of locations matching the keyword and the criteria.

Step 9: The returned path is in the form of a list of edges instead of a list of vertices is due to technical considerations of reroute service

Step 11, 12: Mapbox Map Matching service snaps fuzzy, inaccurate GPS coordinates to the OpenStreetMap road and path network. This produces cleaner paths that can be displayed on a map.

Step 13: In order to draw a path on the map view of Mapbox SDK, the path must be represented as a polyline string.
Reroute Service

Reroute Service is automatically invoked for every X minutes, where X is a constant to be determined. 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.
As shown in Figure 7, the normal workflow is as follows:

1. Frontend service gets the current GPS location
2. Frontend service retrieves the stored path (see Routing Service)
3. Frontend service sends a HTTP request to the backend service with the GPS location as the origin, the destination previously input in Routing service and the retrieved path (the “old path”)
4. Backend service makes a function call to the implementation of the algorithm with the input origin and destination as arguments
5. The implementation of the algorithm returns a new path
6. Backend service compares the estimated traveling time of the old path and that of the new path
   a. Return the old path to frontend service if the estimated traveling time of the old path is not longer than that of the new path for N minutes
   b. Return the new path to frontend service if otherwise
7. Frontend service presents the path to the user
8. The implementation of the algorithm goes through update stage
The detailed backend workflow of the reroute API endpoint is as follows:

1. **APIController** receives the HTTP request from the user with the current position, the destination and the old path as parameters.
2. **APIController** calls the `mapGeoLocation` function of **RoutingKit**. 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 8, the backend workflow of the reroute endpoint is unfolded as follows:

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

**Figure 8: Reroute service backend sequence diagram**
3. APIController calls the clearDensityImpact function of UpdateModule 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 APIController calling the route function in the RoutingModule.

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. APIController then calls updateDensity function of UpdateModule to perform updates in the Current Density cache with the adopted path.

**Frontend**

This service is transparent to users. Frontend will automatically trigger this service for every X minutes, where X is a constant to be determined, until users have reached the destination or have closed the application.
Incident Management

There will be 2 API endpoints for the report of the appearance of incidents as well as the resolution of incident accessible by the external incident reporter. Details are unfolded as follows:

Incident Appearance

![Sequence Diagram](image-url)

*Figure 9: Incident Appearance handling backend sequence diagram*
Incident Management System - Interim Report

Figure 9 shows the flow of handling an incident report in the backend:

1. APIController receives the incident location and the predicted impact, then performs Reverse Geocoding to find the nearest edge and get the corresponding edge ID.
2. APIController creates an entry in Incidents with the maintained incident ID and the predicted impact.
3. APIController adds the incident ID to the set of incidents of the affected edge in AffectedRoads. If there is no entry for the road, create one and include the incident.
4. 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 Resolution.

Incident Resolution

![Diagram](image)

Figure 10: Incident Resolution handling backend sequence diagram

Figure 10 shows the flow of handling an incident resolution report in the backend:

1. APIController receives the incident ID.
2. APIController removes the entry from the Incidents hashmap.
3. APIController searches for the incident ID in all the values (incident sets) of AffectedRoads, remove the incident if it is found, remove the whole key-value pair if the incident set is empty.
4. APIController returns success to the incident reporter.
Simulation

In addition to the end user functions, this application also serves as a graphical simulator. Although the various components of the system can be implemented as a working application without the simulator. However, in order to have a robust and structured system, having the ability to test various scenarios and display the result graphically without making changes to the rest of the system is important.

As shown in Figure 11, the workflow of the simulation is simple:

1. User chooses the type of simulation to conduct
2. The application generates a list of origins and destinations based on the selected type
3. For each pair of origin and destination
   a. Treat it as ordinary O.D. request and run Route Service

There are 5 types of simulation, which each of them indicates how the origins and destinations are generated.

For the first 4 types, the coordinates are selected from a fixed set of 100 coordinates.

1. Single Origin to Single Destination Simulation
2. Single Origin to Multiple Destinations Simulation
3. Multiple Origins to Single Destination Simulation
4. Multiple Origins to Multiple Destinations Simulation
These 4 simulations are enough covered all possible cases. However, it is important to make sure the algorithm not only works on the coordinates presets, but also arbitrary locations. Therefore, an extra type of simulation is included.

For the last type of simulation, the coordinates are generated by pseudo-random number generator bounded within the HKSAR map boundary (see Figure 12). The reason for using a bound polygon instead of a simple bound box is because the surface area of water is significant when comparing to that of land. Using a rectangle boundary will generate a significant amount of coordinates pointing towards the water, causing too few successful cases (i.e. cases where there exists at least 1 path which can travel from the given origin to the destination) being generated.

On the other hand, although the current boundary is imperfect, which some of the water regions like lakes are not excluded and some land regions like Chek Lap Kok are not included, the number of successful cases are enough and the boundary serves its purpose for this project. While the failure cases (i.e. cases where there is no possible path traveling from the given origin to the destination) can be served as part of the simulation. In that case, there is no need to generate failure cases for testing intentionally.

Figure 12: HKSAR boundary (red line)
Current Progress

Mobile Application

One of the design principles of the application is the UI should be similar to that of popular map applications while making necessary changes to adopt our project (See Figure 15). This is because users should be familiar to the operations, which can shorten the learning curve of using this application.

Home page

Being the first page displayed when executing the application, this page features a Map View implemented using Mapbox SDK, a Search Box at the top of the screen and two buttons placed at the bottom right corner of the screen (see Figure 13).

The first button is the Location Button, which enables users to display or hide their current location on the map. When users choose to display their current location, a dialog will be prompted asking for permission to turn on the location service (GPS) of their device (see Figure 14). The second button is the Direction or Route Planning Button, which navigates users to the Directions Page and enables users to get the directions.
Search Locations

Similar to most of the map services, this application features a location searching function. It relies on the Mapbox Geocoding API to return a list of locations which match the user's input. However, this API is not perfect for Hong Kong usage. For example, when the keyword is “HKU”, the locations returned is mostly related to Hong Kong University of Science and Technology (HKUST), while The University of Hong Kong is not even on the list. That being said, locations searching is merely a tool, but not the main focus of this project. Therefore, it is acceptable for using a suboptimal service.

For the UI design, instead of displaying the locations in a list of texts, a list of cards featuring a photo and a brief description (if available) for each location is displayed. This is because it is not uncommon to see multiple locations share the same name. For example, apart from the well-known Queen’s Road in Hong Kong, there is another Queen’s Road located in India as well as a “Queens Road” in London. With the addition of the photos and descriptions, it is easier for the users to distinguish the locations.

Moreover, the search result is in favour of the locations that are closer to user’s current location. That means if a user is searching for “Queen’s road” in Hong Kong, Queen’s Road, Hong Kong would be prioritized before Queen’s Road, India. This further speed up the process of location searching.
Route Planning

Being the most important part of the application, the Route Planning Page features a list displays and allow users to input the desired starting point and destinations (See Figure 17 (a)). The application will send the selected starting points and one or more destinations to the server. After running the algorithm, the server will report a suggested path to the application. The application then will display the route on the map.

For the UI design, the waypoint list is displayed in a separated page instead of at the top of the screen together with the map (See Figure 17 (b) and Figure 18). This is because this application is expected to be mainly used in smartphones which have small screens. When multiple stops and destinations are selected, the list will occupy a significant amount of space, blocking the map.

Moreover, the “add stop” function has been visualized using a large button placed at the bottom of the screen, instead of hidden as a menu item in a drop-down menu (See Figure 19). This is because “add stop” is considered as a primary action and expected to be used frequently. Therefore, it is placed in a more accessible manner.
As mentioned, a simulator has been developed for testing and demonstration purposes. As shown in Figure 20, a drawer menu has been developed to allow users to choose the type of simulations and adjust the parameters respectively. For the 4 preset simulations, exactly 100 O.D. requests will be generated. While for random simulation, users have the control over the number of requests generated range from 1 - 100.
Backend Server

Currently, a few major components of the backend server are implemented for the testing purpose of the frontend and the convenience of the next steps of implementation. These components include the routing API endpoint and the MapGraph Serializer applet.

Routing API Endpoint

The routing API Endpoint is now implemented with temporary functionalities. It now mostly follows the workflow discussed in Figure 7 by randomly picking a path from 100 pre-defined routes to be returned to the client after finding the nearest nodes for origin and destination in the map graph. Moving forward, the current version of this endpoint implementation can be mostly retained as only the random-path-picking function is needed to be replaced by the real routing implementation provided by the RoutingModule. In other words, the main structure of the endpoint has been established with the current implementation.

Figure 22 shows a sample response that a user client may get by sending a POST request to the /route API endpoint. Notice that the paths received may differ from time to time according to the random result generated in the backend.

Besides, the data returned now are contains only the path itself represented as an array of coordinates of each node for the first stage of Frontend application testing. However, in the next stage where rerouting functionalities are incorporated, the data included in the response will include the edges involved in the shortest path instead of the nodes, as well as the time information of when the vehicle enters each edge and when the shortest path starts and ends. This change is needed to facilitate the rerouting function in the case where the backend server does not have any cache or information of the returned paths.
Challenges and Limitations

Failed attempts on algorithm design

The algorithm proposed in the methodology has undergone multiple attempts before arriving at the current version. The failed attempts are mostly caused by either unfamiliar aspects over the problem domain, oversight on the problem, or overlooked boundary cases that render the attempted algorithms flawed. This costs us a fair amount of time and effort that in turn can be used on other parts of this project. The design of the current version of the algorithm we have come up with, is more sound and analysis shown positive results. It takes a less aggressive approach than the previous attempts, and also reuses certain parts of the algorithm from the previous attempts, making the past works done not so much in vain.

Testing on iOS

React Native application can be compiled into an iOS application. However, iOS application can only be compiled and tested using Apple devices, and mostly with Xcode IDE. Yet, none of the members own an Apple mobile device and only one member owns an Apple computer. Therefore, a macOS virtual machine will be created for testing iOS application. Since cross-platform applications are not the main focus of this project, therefore this project will tentatively focus on testing in Android, which the support of iOS will be in lower priority.
Future Plans

Algorithm

Now that the basis of the algorithms used in the system has been formally established, the next step should be proofs of various aspects of the algorithms such as correctness, optimality, complexity, etc. This task is allocated with the highest priority as it determines the successfulness of the whole system.

Backend Server

After the routing algorithm is proved, the implementation of the routing algorithm has to be done to provide the end-to-end functionality of the system. Now that the structure of the routing API is established, it is convenient to plug in the RoutingModule implementation. On top of the routing algorithm, the other endpoints and functionalities like Current Density update, Incident Appearance and Incident Resolution also need to be implemented as soon as possible.

Frontend Application

Since new endpoints will be added and changes to the existing endpoints will be made to the backend service, the frontend application has to adjust the HTTP request handler accordingly to match the new set of endpoints.
Conclusions

We have presented a framework aimed to provide routing service while taking traffic incidents and congestion in Hong Kong into consideration. This application demonstrates the idea of traffic management and describes the implementation and methodology of the system. Being able to support dynamic graph is what makes this project special. Most of the research and studies are focusing on static graphs and leave limited room for the use of real-time data.

Going forward, more effort is required in the service implementation and the algorithm design part for establishing the corresponding proof. Currently this project has proved the lower bound of the algorithm is admissible, while more effort has to be made to form a more comprehensive proof about the application of the heuristic with the A* algorithm itself. Besides, effort is also required in the implementation of the routing algorithm modules and the other endpoints so that the service can be tested and demonstrated.
References


