Incident Management System
For Intelligent City Development

April 14, 2019

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

Written by Yuen Hoi Man
Abstract

The congestion problem is one of the most serious problems within the realm of traffic management in large cities such as Hong Kong. This project aims at finding a solution for the congestion problem by creating a shortest traveling time route suggestion service for Hong Kong. Beside taking incident into consideration, the routing service also aims to be aware of the congestions caused by previous routed request. Although shortest path routing in road networks is a well-studied problem and many different search methods that can provide a decent speedup from the traditional Dijkstra’s Algorithm exist, none of the methods can be directly applied to the routing service this project aims to create.

This report introduces a new speedup method that is sensitive to small changes within the road network, which is able to handle incidents and the small traveling time increases caused by previous routings. The algorithm takes root from the A* algorithm, taking advantage of multi-layered partitioning of the original road network, and created precomputed data in order to achieve average speedups of 13 against the traditional Dijkstra’s Algorithm when working on a dynamic network. Tests and experiments were done on the implementation of the algorithm in order to determine the speedup, the performance and also the recommended parameters for use in the routing service. Finally, some suggestions are also provided for using the algorithm and the service in a different place than Hong Kong.
Acknowledgments

We would like to thank our supervisor, Dr. Cheng, for guiding us throughout the project. We are also grateful for the helpful advice given by Professor Kao that are really useful when working on the project.
Table of Contents

Abstract 1
Acknowledgments 2
Table of Contents 3
List of Figures 4
List of Tables 4
Abbreviations 5

1 Introduction 6
   1.1 Background and Motivation 6
   1.2 Objective and Scope 8
   1.3 Deliverables 9
   1.4 Outline of Reports 10

2 Methodology 11
   2.1 Overview 11
   2.2 Related Works 12
   2.3 Layered Partitioning 15
   2.4 Preprocessing 17
   2.6 Query 21

3 Experiments 25
   3.1 Setup 25
   3.2 Efficiency 26
   3.3 Storage space 33
   3.4 Result Discussion and Suggestions 34

4 Difficulties encountered 35
   4.1 Different iterations of algorithm designs 35
   4.2 Implementation of the algorithm 35
   4.3 Testing of the algorithm 36

5 Conclusion 37

References 38
List of Figures

Figure 1  An example of congestion induced by the routing service itself 7
Figure 2  Apply layered partitioning to a graph $G$ with $k = 3$ 16
Figure 3  The temporary vertices added and distances obtained during preprocessing 18
Figure 4  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 22
Figure 5  An illustration of partitioning the Hong Kong graph with a simple rectangular grid partition method where $k = 4$ 23
Figure 6  Expanded nodes for varying parameter settings on SS-HKU requests 28
Figure 7  Expanded nodes for varying parameter settings on SS-HKU requests 28
Figure 8  Search space of SS-HKU test requests 30
Figure 9  Search space of KT-TM test requests 30
Figure 10  Search Space of TaW-ST test requests 31
Figure 11  Search Space of TsW-MK test requests with approximated partition size for $k = 32$ 31
Figure 12  Search Space of TsK-KC test requests for $k = 32$ 32
Figure 13  Search Space of TsW-KC test requests for $k = 64$ 32
Figure 14  Fitting storage size on parameter $k$ 33
Figure 15  Fitting storage size on parameter $l$ 33

List of Tables

Table 1  Performance summary on SS-HKU test requests 27
Table 2  Performance summary on all sample requests 29
Table 3  Storage space needed for each precomputed data (bytes) 33
## Abbreviations

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALT</td>
<td>A* search on Landmark nodes using Triangle inequality</td>
</tr>
<tr>
<td>CH</td>
<td>Contraction Hierarchies</td>
</tr>
<tr>
<td>IMS</td>
<td>Incident Management System</td>
</tr>
<tr>
<td>KT-TM</td>
<td>Kwun Tong - Tuen Mun test routes</td>
</tr>
<tr>
<td>PCD</td>
<td>Precomputed Cluster Distances</td>
</tr>
<tr>
<td>SS-HKU</td>
<td>Sheung Shui - University of Hong Kong test routes</td>
</tr>
<tr>
<td>TaW-ST</td>
<td>Tai Wai - Sha Tin test routes</td>
</tr>
<tr>
<td>TsW-KC</td>
<td>Tsuen Wan - Kowloon City test routes</td>
</tr>
<tr>
<td>TsW-MK</td>
<td>Tsuen Wan - Mong Kok test routes</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background and Motivation

Traffic management is one of the most crucial problems in cities such as Hong Kong where traffic efficiency and quality is highly important among the dense population. But unfortunately, the problem concerning traffic efficiency and quality itself is undoubtedly vague and complex. It is not possible to handle the entirety of the problem as a whole, and should better be divided into concrete sub-problems that can be more easily identified and solved. In respect to within only our expertise, there are a few sub-problems that can be solved. The first and foremost sub-problem we would like to tackle is routing.

Real-life road networks tend to be mostly vast and disordered, causing drivers unable to easily determine the best path at a glance. Popular solutions such as Contraction Hierarchies[1] and Customizable Route Planning[2] models the routing problem as a shortest path search on a positive directed graph, and can quickly identify the path with the shortest travelling time between any two points in the road network, thus provide drivers the correct direction and let them concentrate on driving. These solutions mostly work in a similar fashion, which usually involves a long, and usually resources expensive, preprocess period on the road network, retrieving relevant information that may speed up the routing process. The solutions, however, usually assumes the road network is static to a certain degree in order to work efficiently. This is mostly not the case in many real-life scenarios, as there will be occasions that some proportions of the road network are congested by traffic incidents, events, and other unexpected situations, that the traveling time will be affected for those road segments. Handling congestions is an important topic of traffic management, as in the case of Hong Kong, citizens suffer at least a 200% increase in commuting time by congestions caused by traffic jams [3]. This poses the second sub-problem from traffic management that we would like to solve. The above-mentioned solutions usually do not handle these changes within the algorithm directly, but rather re-trigger the processing phase in order to take those changes into consideration. This causes them losing their advantages when congestion happens frequently, where triggering the expensive preprocessing phase every time some major change occurs is not efficiently at all.

The final sub-problem we would like to tackle is the side effect of a routing service. Given a scenario, such as after a large public event, that a large number of drivers uses a single routing service. It is possible that much of the drivers will be given a path via same road segments at about the same time, and thus the routing service itself may cause congestion at that road segment. This would result in providing the driver a suboptimal path, as well as a general impact on the overall traffic quality. Most of the current solutions do not handle this situation. We speculated that this problem may not pose a significant impact in current road

6
network where manual driving is still dominant, but nevertheless would be quite useful in a situation where the whole traffic system is centrally automated in the future.

Figure 1: An example of congestion induced by the routing service itself

Less traffic efficiency leads to a huge waste of valuable time and even economic loss since less time could spend on economically productive activities[3]. Along with the increase in the total number of vehicles at least about a hundred thousand[4], we felt that a traveling route suggestion service that can take both incident and previous routes into consideration may provide a way to alleviate the problem. Currently, amongst all available solutions for this problem, TallyGo[5] (former ClearPath) provides the most similar service as we would like to deliver. Despite being a well-established product, there are a few shortcomings in TallyGo that we think it may not be sufficient. The reasons being firstly, 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. And secondly, TallyGo does not specifically handle the impact from large-volume requests, which as stated above, few or even none of the current traffic solutions solve.
1.2 Objective and Scope

The objective of the project is to create a route planning service for traffic management in Hong Kong that can detect and react to different real-time traffic scenarios autonomously. Due to the wide breadth of the two components and also time constraints, we further limit the scope of this project such that it focuses on taking real-time events into consideration while doing shortest path searches on the road network.

From the limitations stated above, we aim to deliver a web-based incident management service (IMS) that provides route suggestion functions focusing on the two main contributing factors of traffic congestions, namely large-scale events and road incident, as well as fill in the gap left by the other services. For large-scale events such as the Hong Kong Book Fair and large concerts held by famous singers, huge traffic flow and thus extra congestion may be generated by a large number of participants of those events. Our service aims to suggest efficient routes that minimize extra congestion created, ensuring a reasonable traveling time for the user. For road incidents, in order to avoid duplication of work, since the upstream incident detection mechanism relying on traffic data provided by Hong Kong taxis is currently being handled by a peer group under the same supervisor Dr. Cheng, this project and thus the service will put focus on using existing Hong Kong road map data for the routing requests assuming incident detection is available for access. The service tries to suggest a route which has minimum path segments passing through any congested area affected by incidents.

Along with the service implementation, this project also tries to deliver a set of algorithms that are sensible of changes in edge weight in the road network, both from live incident and other requests, that can provide a shortest path routing respond in a reasonably short amount of time even for intensive online usage.
1.3 Deliverables

There are a total of 3 major deliverables in this project:

1. Routing algorithm:
   A set of algorithms designed to solve origin to destination routing request that takes real-time events and previous routing requests 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 each routing request) and pass them to the corresponding algorithm modules for the results needed to be returned to the Frontend.

3. Frontend service:
   A mobile application that acts as both the application prototype and simulator, the main entry point for any user to access the service.
   a. Application prototype provides a routing planning service. It also includes basic features that were expected to exist in an 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.
1.4 Outline of Reports

The documentation of for this project is divided into three separate reports, where each report emphasizes on the methodology, experiments, and results, and difficulties encountered when working on different distinct parts of the project. It is strongly advised for readers to refer to each individual reports in order to gain a full picture of the project. A brief summary of the three reports are as follows:

1. Report on Frontend service:
   Written by Kwong Matthew Wang Shun, focuses on the full design and implementation of the frontend service, includes both the application prototype and the simulator.

2. Report on edge update algorithm and Backend service:
   Written by Chow Yin Chak, highlight the details of the algorithm on feeding the impact from previous routed requests into the system, the management of said impacts, as well as the design and implementation of the Backend service.

3. This report:
   Outline the design of the routing algorithm used by the service. Firstly, the report gives an overview of the algorithm we desire, and then inspects currently available algorithms that solve shortest path search on a road network and highlight the reason they cannot be used directly in our service, and thus leads to the design of our algorithm. Afterward, implementation of the algorithm and experiments done are discussed, along with some recommendation for using the algorithm. Finally, the report ends with some conclusion for the whole project.
2 Methodology

2.1 Overview

The algorithms being used in the Incident Management System can be roughly divided into two parts, which are the routing algorithm and the edge 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 edge 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. In order to support the functionality of the edge update algorithm, the routing algorithm should allow high-frequency edge updates, as well as high sensitivity to the weight changes made on the road network. From these criteria we propose the use of the A* algorithm[6] 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 edge update according to an accepted and returned path and according to time-independent incidents are also proposed. By the linear relationship between traveling speed and traffic density, the updates in edge weights (traveling 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. 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. Please be noted that the details for the edge update algorithm is out of scope for this report, but is the responsibility of the report on edge update algorithm and Backend service written by Chow Yin Chak.
2.2 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[7], the problem is considered to be solved in a relatively optimal manner. But still in some real-life applications such as road network pathfinding, 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[8], Contraction Hierarchies(CH)[1], Multi-Level Dijkstra[9], Customizable Route Planning[2], Precomputed Clustered Distances(PCD)[10] and A* search on Landmark nodes using Triangle inequality (ALT [11] provided huge speed ups to Dijkstra’s algorithm in a way that real-time shortest path searches are possible even in a huge road network.

To briefly summary how speedups can be achieved on the already optimal Dijkstra’s Algorithm[7], all speedup techniques either exploit all or at least some of the following properties of a road network routing:

1. Road networks are relatively stable such that the hierarchy of the network does not change frequently (roads are a quite static object in the real world)
2. Shortest path search in road networks are most likely to be informed search (information are available for the destination)
3. Road network routing used in services mostly require routing on the same road network only but in an excessive number of times (all searches using the same service are most likely to be riding on the same road network)
4. There exist some main road segments such that it facilitate most of the shortest paths (motorway, expressway, Autobahn, etc.)

All of the algorithms mentioned made their speedup of the shortest path search by extracting and condensing information of the graph during a relative long preprocessing phrase, long before any actual query is done. This effectively shifts part of the time the calculation needed during short path search to the preprocessing phase, which when querying is needed, the information acquired can be used to prune or prioritize individual edges on the road network. By such a method, it is possible to break the lower bound of the complexity established by Dijkstra’s Algorithm when only querying is concerned. Among all the algorithm stated above, CH yields an impressive speedup by extremely assigning priority to every single edge within the road network and prune edges that does not appear in any shortest path between any two edges, in the cost of a much longer preprocessing time needed.

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][2], or predictable changes at best[10] in order to function properly. One of the ways that modern system do to handle huge changes within real-life road network is to redo precomputation per a set amount of time. This normally would be sufficient in handling road incidents, but do not fit our purpose of reducing impact from previous routed requests. This is caused by the weight changes induced by previous routed requests tends to be timed and short, which is impossible (the weight change caused by previous routes is time-dependent, and will not be reflected in a static graph) and inefficient (redo resource expensive recomputed multiple times within a short period) to be captured using the above method. The limitation of our system renders most of the speedup techniques not suitable for our situation. It is also important to note that since most of the mentioned speedup techniques utilized precomputation to facilitate the speedups, 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[6], 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[12], 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[13], 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.

One of the algorithms mentioned above does use the A* algorithm[6] as the basis of their design. It also allows using dynamic road network instead of static road network when doing computation, which is ALT[11]. ALT does not prune any edges on the road network, which is essential for shortest path search in a dynamic network where edges are changing. Then, ALT selects a number of landmark nodes such that the shortest path distance between each
landmark is preprocessed. This information is then used to provide the lower bound used in the search when performing queries. It is stated that the selection of the landmark nodes is an extremely important contributing factor of the speedup ALT can provide, where landmark nodes are usually nodes stated in property 4. This poses a problem for our system, as since we know motorways are usually where traffic incidents and congestions take place, and especially the case in Hong Kong, it is not a good idea to rely on property 4 for the speedup.

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[2], this definition should be sufficient for general purpose pathfinding for typical usage. Our algorithm takes inspiration for multiple speedup methods stated above, which are Multi-Level Dijkstra[9] for the idea of level-of-detail of information, PCD[10] for grouping nodes, and finally ALT[11] for ways of utilizing A* Algorithm in handling dynamic graphs.
2.3  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 $c_i$. $C_{i+1}$ 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:

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

The full partitioning process terminates at level $l_{\text{max}}$ 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 $l_{\text{max}}$, it provides little to none new useful information in regards to calculating the lower bounds. Furthermore, although it may is beneficial to achieve partitions up until level $l_{\text{max}}$, in practical cases we should select a parameter $l \leq l_{\text{max}}$ such that the precomputed data generated during preprocessing would take up a reasonable size of storage spaces. The selection of the partition method used within layered partitioning will
affect the final efficiency of the routing algorithm, but it would discuss later within this report. Fig. 1 illustrates an example of the layered partitioning process and the resultant layers from the process.

![Layered Partitioning Example](image)

**Figure 2: Apply layered partitioning to a graph G with k = 3**

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:     $B_i = B_i \cup \{p(c_{i-1})\}$

2.4 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 traveling time for \( m_i \) to exit the bound that contains itself, as well as the shortest possible traveling time for reaching \( m_j \) 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 \( l \) 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 PCD[10] along with some slight modifications, which can be described as follows:

For each partition \( m_i \) of layer \( L_i \), a temporary vertex \( v_m \) 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 \( m_{i,1} \) at layer \( i-1 \) of bound \( M_i \), where \( M_i \) is the bound containing \( m_i \). A total of three 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 \( v_m \) 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 \( n_i \neq m_i \) that was contained in the same bound as \( m_i \) was reached, or more precisely, one of the vertices from each partition \( n_i \) are reached. The second single source shortest path search is similar to the first, but instead of targeting all partitions \( n_i \), the search targets only the vertex \( v_M \) instead. The search can be prematurely stopped when \( v_M \) is reached. The third single shortest path search originates from \( v_M \) on graph \( G \) that target \( v_m \). Same as the previous search, this search can also be prematurely stopped when \( v_m \) is reached. The shortest path distances from \( v_m \) to all \( n_i \), from \( v_m \) to \( v_M \) and from \( v_M \) to \( v_m \) are recorded as entries for \( d(m, n) \), \( d_m(m) \) and \( d_M(m) \) in distance table for layer \( L_i \). 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.

![Diagram](image)

*Figure 3: The temporary vertices added and distances obtained during preprocessing.*

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 to be 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 \(\Theta(k^{l+2})\) storage space, and can be further shown to take up \(e(\frac{1}{1-k} - k - 1)\) space, where \(e\) is the space needed for storing each entry in the distance table.

**Lemma 1.** The precomputed data takes \(e(\frac{1}{1-k} - k - 1)\) 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
The fact that level 0 should not contain non-trivial information. Level $l$ contains $k$ partitions, thus requires a total of $k^l$ 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

$$e(k^2 + k^3 + \cdots + k^{l+2})$$

$$= e\left(\sum_{i=0}^{l+2} k^i \right)$$

$$= e\left(\frac{1 - k^{l+3}}{1 - k} - 1\right)$$

\[ \square \]

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 preprocessing 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 $L_i = (C_i, B_i)$, $i \leq 1$ 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 $e_m$ be a set of zero weight edges from $v_m$ to each vertex of $m_i'$
8: $V = V \cup \{v_m\}$
9: $E = E \cup e_m$
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 $u$
if $c \in N_i$ and $rep[c] \neq \text{Nil}$

$rep[c] = u$

if $\forall n_i \in N_i, rep[n_i] \neq \text{Nil}$

break

for each $v \in \text{neighbors of } u$

if $\text{dist}[v] > \text{dist}[u] + w(u, v)$

$\text{dist}[v] = \text{dist}[u] + w(u, v)$

$\text{prev}[v] = u$

for each $n_i \in N_i$

add $\text{dist}[rep[n_i]]$ as an entry for $d(m_i, n_i)$ to $DT_i$

let $v_M$ be a new temporary vertex

let $e_M$ be a set of zero weight edges from each border vertices of $m_i$ to $v_M$

$V = V \cup \{v_M\}$

$E = E \cup e_M$

add the shortest path distance from $v_m$ to $v_M$ as an entry for $d(m_i, n_i)$ to $DT_i$ (36)

add the shortest path distance from $v_M$ to $v_m$ as an entry for $d_{\text{in}}(m_i)$ to $DT_i$

$V = V \setminus \{v_m, v_M\}$

$E = E \setminus e_m \cup e_M$

Ideally, if the road network itself does not encounter any significant 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.
2.6 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, querying 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 \( h(u, t) \) 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 \( h(u, t) \) 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:

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

where \( d(U_i, T_i), d_{out}(U_i), d_{in}(T_i) \) 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.** \( h(u, t) \leq h^*(u, t) \)

The aim of this proof is to show that the lower bound \( h(u, t) \) we suggested do not overestimate the actual cost \( h^*(u, t) \) 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 \((u, \ldots, t \in V)\) from any vertex \(u\) to any vertex \(t\), \(U_i\) 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 \((u, \ldots, u'), (u'_1, \ldots, u'_{i-1}), (u'_{i+1}, \ldots, u'_m), (u'_m, \ldots, t'_m), (t'_m, \ldots, t'_{m+1}), \ldots, (t'_{i-1}, \ldots, t'), (t'_p, \ldots, t), \) where \(u'_j\) denotes the first vertex the path reach that belongs to a border vertex of \(U_i\) and \(t'_j\) denotes the last vertex the path reach that belongs to a border vertex of \(T_i\). 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, \(h^*(u, t)\) can be split into multiple parts corresponding to each sub-path such that \(h^*(u, t) = h^*(u, u'_j) + h^*(u'_j, u'_{i-1}) + \ldots + h^*(u'_{m+1}, u'_m) + h^*(u'_m, t'_m) + h^*(t'_m, t'_{m+1}) + \ldots + h^*(t'_{i-1}, t') + h^*(t'_p, t)\).

Expanding \(h(u, t)\) by the definition yields the expression \(h(u, t) = d_{out}(U_1) + \ldots + d_{out}(U_m) + d(U_m, T_m) + d_{in}(T_m) + \ldots + d_{in}(T_i)\). Fig.3 shows an illustrated example of the splitting of \(h^*(u, t)\) and the expansion of \(h(u, t)\) on the same graph.

![Figure 4: 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 \(i\).](image)

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

\[
\begin{align*}
    h(u, t) &= 0 + d_{out}(U_1) + \ldots + d_{out}(U_m) + d(U_m, T_m) + d_{in}(T_m) + \ldots + d_{in}(T_i) + 0 \\
    h^*(u, t) &= h^*(u, u'_1) + h^*(u'_1, u'_{i-1}) + \ldots + h^*(u'_{m+1}, u'_m) + h^*(u'_m, t'_m) + h^*(t'_m, t'_{m+1}) + \ldots + h^*(t'_{i-1}, t') + h^*(t'_p, t)
\end{align*}
\]

that either the comparisons are trivial or must be true since \(d(v_1, v_2)\), \(d_{out}(v)\) 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 \(h(u, t)\) at all. Therefore, \(h(u, t) \leq h^*(u, t)\) must hold. □

22
With this definition of the lower bound function, two effects can be observed as for 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. This is mainly because any precomputed distance information across levels is only encapsulated by \( d_{in} \) and \( d_{out} \) for each partition. 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 by bound, it can be concluded that in order for this lower bound function to work more efficiently, the partition criteria should focus on dividing the vertices into many partitions, where the number of partitions containing border vertices of the bound should be small. Although the number of layers seems to play a less significant role in the efficiency of the lower bound, it nevertheless can provide a sense of direction when the search getting close to the target.

Given by the information above, it is clear that the choice of partition methods this project used earlier during layered partitioning have some certain impacts on the efficiency of the routing algorithm. On the other hand though, given any partition methods, the algorithm still remains optimal as the lower bound function is still admissible. This claim is supported by Lemma 2, from which the lower bound is already shown to be admissible even when independent of the partition method used to create the distance tables. Due to time constraint, we employ a simple rectangular grid partition method which simply divides the whole graph into grids of \( k \times k \) partitions according to their geographic location. We are well aware that there are other partition methods such as k-center clustering with either MinSize or MinRad heuristics[10], that would most likely to yield better results than our simple square grid partitions, but latter would serve as a good baseline for the performance of our routing algorithm during experiments.

![Figure 5: An illustration of partitioning the Hong Kong graph with a simple rectangular grid partition method where \( k = 4 \) ](image-url)
Finally, the current cost function \( g(s, u) \) for calculating total traveling 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) traveling 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 design of the time-dependent edge weight retrieval function falls into the scope of another report and will not be discussed here. In short, the time-dependent edge weight retrieval function \( w'(e, t) \) is given by the following formula

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

Where \( w(e) \) is the weight function of an edge \( e \), \( \rho(e, t) \) and \( \rho_0(e, t) \) are density functions at a particular \( t \) on edge \( e \), \( a(e) \) is a time-independent modifier function of an edge \( e \). Both the impact from incidents and previous routed requests are encapsulated within these functions. It can also be proved that \( w'(e, t) > w(e) \) in every case, such that our lower bound remains admissible.
3 Experiments

The implementations of the routing algorithm used in the Incident Management System is fully completed, and experiments and studies have been done on the routing algorithm to understand the performance of it, and also determine the practical configuration for usage as a part of the Incident Management System.

3.1 Setup

As the Incident Management System aims at creating a service for Hong Kong, it is natural for one to perform experiments on the information of Hong Kong. During the course of this experiment, a data set of the road network of Hong Kong was retrieved from OpenStreetMap[14], a community-driven mapping project, where the data set we needed can be obtained via their directory[15]. For the purpose of this experiment, we assume the map data is correct in a way that the final results when deploying the service do not differ from the experimental result too much. A road network graph was then extracted from the retrieved map data by Chow and the details from the extraction will be discussed in the report of edge update and backend service.

The road network graph was preprocessed in the following way, from which we would later study the different aspects of the routing algorithm. For the parameter of \((k, l)\), where \(k^2\) is the number of partitions generated per partition method, and \(l\) is the total number of layers we partition the graph into, we preprocessed graphs for \(k = 4, 8, 16, 32, 64\), and \(l = 2, 3, 4\). Due to computational and time constraint, data for \(l = 5\) were not being considered as these both take too much time to make, and are too big for storage, but we still manage to get an additional data of \((8, 5)\) for testing. In order to compare the results, a traditional Dijkstra’s Algorithm was also performed. Because of the fundamental differences between our algorithm and most of the other speedup algorithms, differences such as dynamic network and time-dependent weight change, it is from difficult to impossible for us the fairly compare the algorithms. This phenomenon can also be observed in the experimentation of PCD[10], where only Bidirectional Dijkstra is available for us to use in comparing the performance. Unfortunately, in our case, even Bidirectional Dijkstra is not possible due to the fact that weight change is time dependent and cannot be determined beforehand.

The routing algorithm itself was implemented in C++, and all tests were performed on a virtual machine with a LINUX operating system. The virtual machine was configured with a single processor and with 4GB of RAM. Please be noted that this hardware configuration has significantly lower processing power when compared to those used in deployment of the service, such that all hardware dependent information are just here for reference only.
3.2 Efficiency

Five origin-to-destination routing requests were hand-picked to test how the algorithm performs. Each request picked have real-life implications and possess some special properties that certain strengths and weaknesses can be identified through studying the results. The five requests with their properties (p) and real-life implications (i) are listed as follows. For the sake of ease of reading, the term “route” used here refers to the path with the shortest traveling time between the origin and destination in the real world.

1. Sheung Shui - University of Hong Kong (SS-HKU)
   p1. This route is longest among the five requests.
   p2. This route spans almost the entire diameter of the Hong Kong road network.
   p3. Almost the entire route utilizes motorways with the exception of near the start and end of the route.
  i1. This request represents one type of general commuting for citizens living in the New Territories to somewhere on Hong Kong Island.

2. Tai Wai - Sha Tin (TaW-ST)
   p1. This route is shortest among the five requests.
   p2. This route confines within a small local area without traveling outwards to other areas.
   p3. This entire route does not utilize any motorway.
  i1. This request represents local travels within the same town for everyday activities.

3. Kwun Tong - Tuen Mun (KT-TM)
   p1. This route spans around half the diameter of the Hong Kong road network.
   p2. This route utilizes a fair amount of both motorway and city roads.
  i1. This request represents another type of general commuting for citizens living in the New Territories to somewhere on the Kowloon.

4. Tsuen Wan - Mong Kok (TsW-MK)
   p1. This route confines within a small but highly connected area.
   p2. This entire route does not utilize any motorway.
  i1. This request represents travels within the city area usually with slow traveling speed.

5. Tsuen Wan - Kowloon City (TsW-KC)
   p1. The route is a mutation of TsW-MK that ends in a highly private area.
   p2. This request highlight a certain weakness of the algorithm.
Firstly we inspect the information retrieved from running the SS-HKU test requests. Table 1 gives a summary of the performance of the algorithm using all preprocessed data we have acquired.

Table 1. Performance summary on SS-HKU test requests

<table>
<thead>
<tr>
<th>Parameter (k-l)</th>
<th>Initial time (ms)</th>
<th>Routing time (ms)</th>
<th>Expanded nodes</th>
<th>Expanded edge</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dijkstra</td>
<td>400</td>
<td>1157</td>
<td>30267</td>
<td>31336</td>
</tr>
<tr>
<td>4-2</td>
<td>411</td>
<td>1175</td>
<td>30267</td>
<td>31336</td>
</tr>
<tr>
<td>8-2</td>
<td>402</td>
<td>1139</td>
<td>30267</td>
<td>31336</td>
</tr>
<tr>
<td>16-2</td>
<td>398</td>
<td>1152</td>
<td>30267</td>
<td>31336</td>
</tr>
<tr>
<td>32-2</td>
<td>401</td>
<td>1146</td>
<td>30267</td>
<td>31336</td>
</tr>
<tr>
<td>64-2</td>
<td>408</td>
<td>1139</td>
<td>30267</td>
<td>31336</td>
</tr>
<tr>
<td>4-3</td>
<td>410</td>
<td>1140</td>
<td>30213</td>
<td>31281</td>
</tr>
<tr>
<td>8-3</td>
<td>393</td>
<td>1062</td>
<td>25922</td>
<td>29199</td>
</tr>
<tr>
<td>16-3</td>
<td>413</td>
<td>754</td>
<td>18562</td>
<td>20207</td>
</tr>
<tr>
<td>32-3</td>
<td>440</td>
<td>389</td>
<td>8330</td>
<td>10000</td>
</tr>
<tr>
<td>64-3</td>
<td>1657</td>
<td>82</td>
<td>1486</td>
<td>1530</td>
</tr>
<tr>
<td>4-4</td>
<td>398</td>
<td>1117</td>
<td>28356</td>
<td>30955</td>
</tr>
<tr>
<td>8-4</td>
<td>433</td>
<td>757</td>
<td>17113</td>
<td>20433</td>
</tr>
<tr>
<td>16-4</td>
<td>1074</td>
<td>424</td>
<td>9391</td>
<td>10258</td>
</tr>
<tr>
<td>32-4</td>
<td>9870</td>
<td>108</td>
<td>2095</td>
<td>2290</td>
</tr>
<tr>
<td>64-4</td>
<td>8839</td>
<td>53</td>
<td>1015</td>
<td>1050</td>
</tr>
<tr>
<td>8-5</td>
<td>1021</td>
<td>692</td>
<td>15135</td>
<td>18060</td>
</tr>
</tbody>
</table>

It can be observed from all tests with \( l = 2 \) that the number of expanded nodes and edges are exactly equal to those from the traditional Dijkstra’s Algorithm. This phenomenon can also be observed for all other tests suggest that there are some impurities within the original data itself that, it cause level 1 contains only two partitions where one partition contains most of the graph except one to two nodes. Upon further inspection of the data, we do find one to two nodes with unnatural geographical location (i.e. latitude = 0, longitude = 0). Although this does not affect our routing algorithm in any way since our algorithm does not consider geographical location directly but only the traveling time needed for each edge, this causes the effectiveness of parameter \( l \) lowered by 1. We should take this into consideration when doing analysis on the other information in this experiment.

Another observation made from the above information is that the time needed for routing is linearly proportional to the number of nodes expanded by the algorithm. This is as expected to be the case for all shortest path search algorithms but nevertheless included here to show this indeed comparable to other shortest path search algorithms. Figure 6 shows the plot and also a fitted line for the relationship between expanded edges and the time needed for routing.
A final observation can be seen in figure 7, which we can see the decrease of the number of edges expanded by increasing level is still present, the rate of decrease start to slow down when further increasing the number of levels. This observation has been predicted above, of which we have predicted the importance of increasing the number of partitions (parameter $k$) is more important than the number of levels (parameter $l$). For the routing using $k = 64$, we can see that performance gain from $l = 3$ to $l = 4$ slows down so dramatically, we would argue that this is caused by $k = 64$ is reaching the maximum number of speedups such that performance gain is extremely hard. It should be noted that, although the gain is small, the performance still manage to get a 31% increase in effectiveness from $l = 3$ to $l = 4$, contributing to 1.5% increase of the overall performance.

![Figure 6: Expanded nodes for varying parameter settings on SS-HKU requests](image1)

![Figure 7: Expanded nodes for varying parameter settings on SS-HKU requests](image2)
For a more general understanding of the performance of our algorithm, we would then inspect the information retrieved from running all five previously defined sample requests. Table 2 gives a summary of the information. Since executing time is machine dependent, it poses no significant value when performing analyses. Also, the number of expanded nodes and the number of expanded edges provide similar information, of which, doing analysis on one of them is enough. We further calculate the speedup value, which is defined as the ratio of the complexity of our algorithm and the complexity of the algorithm we are accelerating[10].

Table 2. Performance summary on all sample requests

<table>
<thead>
<tr>
<th>Parameter (k-l)</th>
<th>SS-HKU</th>
<th>TaW-ST</th>
<th>KT-TM</th>
<th>TsW-MK</th>
<th>TsW-KC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Nodes</td>
<td>Speedup</td>
<td>Nodes</td>
<td>Speedup</td>
<td>Nodes</td>
</tr>
<tr>
<td>Dijkstra</td>
<td>31336</td>
<td>1.00</td>
<td>3072</td>
<td>1.00</td>
<td>30896</td>
</tr>
<tr>
<td>4-2</td>
<td>31336</td>
<td>1.00</td>
<td>3072</td>
<td>1.00</td>
<td>30896</td>
</tr>
<tr>
<td>8-2</td>
<td>31336</td>
<td>1.00</td>
<td>3072</td>
<td>1.00</td>
<td>30896</td>
</tr>
<tr>
<td>16-2</td>
<td>31336</td>
<td>1.00</td>
<td>3072</td>
<td>1.00</td>
<td>30896</td>
</tr>
<tr>
<td>32-2</td>
<td>31336</td>
<td>1.00</td>
<td>3072</td>
<td>1.00</td>
<td>30895</td>
</tr>
<tr>
<td>64-2</td>
<td>31336</td>
<td>1.00</td>
<td>3072</td>
<td>1.00</td>
<td>30895</td>
</tr>
<tr>
<td>4-3</td>
<td>31281</td>
<td>1.00</td>
<td>3039</td>
<td>1.01</td>
<td>30880</td>
</tr>
<tr>
<td>8-3</td>
<td>29199</td>
<td>1.07</td>
<td>3039</td>
<td>1.01</td>
<td>21381</td>
</tr>
<tr>
<td>16-3</td>
<td>20207</td>
<td>1.55</td>
<td>2676</td>
<td>1.15</td>
<td>18011</td>
</tr>
<tr>
<td>32-3</td>
<td>10000</td>
<td>3.13</td>
<td>1877</td>
<td>1.64</td>
<td>12036</td>
</tr>
<tr>
<td>64-3</td>
<td>1530</td>
<td>20.48</td>
<td>1590</td>
<td>1.93</td>
<td>4494</td>
</tr>
<tr>
<td>4-4</td>
<td>30955</td>
<td>1.01</td>
<td>3028</td>
<td>1.01</td>
<td>29228</td>
</tr>
<tr>
<td>8-4</td>
<td>20433</td>
<td>1.53</td>
<td>1794</td>
<td>1.71</td>
<td>15923</td>
</tr>
<tr>
<td>16-4</td>
<td>10258</td>
<td>3.05</td>
<td>1479</td>
<td>2.08</td>
<td>7903</td>
</tr>
<tr>
<td>32-4</td>
<td>2290</td>
<td>13.68</td>
<td>386</td>
<td>7.96</td>
<td>3981</td>
</tr>
<tr>
<td>64-4</td>
<td>1050</td>
<td>29.84</td>
<td>372</td>
<td>8.26</td>
<td>2696</td>
</tr>
<tr>
<td>8-5</td>
<td>18060</td>
<td>1.74</td>
<td>1326</td>
<td>2.32</td>
<td>14697</td>
</tr>
</tbody>
</table>

Figure 8 to Figure 11 shows a plot of the expanded edges of each test request with both 32-4 and 64-4 settings, along with the expanded edge with Dijkstra’s algorithm. This two settings is chosen because currently, the setting of l = 4 and especially k = 64 yields the best results from all others, giving an at least 7 times better than using traditional Dijkstra’s algorithm. The algorithm also performs exceptionally well when there is motorway support within the shortest path. This can be seen by both the SS-HKU test requests and the KT-TM test requests. From figure 8 and figure 9, we can see that the expanded spaces around motorways are very tight around the shortest path. This is most likely to be caused by the nodes contributing to the motorway edges provides a good path through a partition, reducing the needed to search much through the partition.
Another observation that we can make from these data, is that the algorithm has a sense of locality reflected in the search space as seen in the TaW-ST test requests from Figure 10 and the TsW-MK test requests from Figure 11. In the case of TaW-ST, the search space of our algorithm do not leave the local area as Dijkstra’s algorithm does even without the help of motorways. This is most likely an effect from the level of details of lower bound given from the layered partitions.
Figure 10: Search Space of TaW-ST test requests

Figure 11: Search Space of TsW-MK test requests with approximated partition size for $k = 32$

Figure 11 gives a more clear demonstration of the benefit of locality brought by a layered partitioning and also shows the importance of parameter $k$ on the overall performance increase. The green square shows approximately the border of each partition when $k = 32$. It is clear that the search space rarely spans beyond the partition border. Within the partition itself, it can also be observed that the algorithm have a slight tendency to push the search towards the bound of the partition, where the slower small streets were not been expanded.
Although in most cases, routing with parameter $k = 64, l = 4$ provides a better speedup than those with $k = 32, l = 4$, there do exist some extreme cases that the latter provides better speedups than the former. These extreme cases are highlighted by the TsK-KC test requests. The request has the same origin as in the TsK-MK test requests but with a destination of a highly private area in a dead end, result in an extremely long traveling time in the final few road segments. This, in turn, causes the algorithm to consider all other edges with the same traveling time. The reason for $k = 64$ performs worse than $k = 32$ is mostly due to the fact that for each node we have a tighter lower bound and thus more nodes to be estimated to have a similar traveling time as the shortest path to the destination.
3.3 Storage space

The storage space needed for the precomputed data of each of the settings can be summarized in the following table.

**Table 3. Storage space needed for each precomputed data (bytes)**

<table>
<thead>
<tr>
<th></th>
<th>$l = 2$</th>
<th>$l = 3$</th>
<th>$l = 4$</th>
<th>$l = 5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$k = 4$</td>
<td>3490972</td>
<td>3496621</td>
<td>3550414</td>
<td>N/A</td>
</tr>
<tr>
<td>$k = 8$</td>
<td>3490972</td>
<td>3546362</td>
<td>3847751</td>
<td>9000487</td>
</tr>
<tr>
<td>$k = 16$</td>
<td>3490972</td>
<td>3568724</td>
<td>9528192</td>
<td>N/A</td>
</tr>
<tr>
<td>$k = 32$</td>
<td>3491038</td>
<td>3928619</td>
<td>83448348</td>
<td>N/A</td>
</tr>
<tr>
<td>$k = 64$</td>
<td>3491114</td>
<td>14289381</td>
<td>78158050</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Again, as observed and identified from the previous part, the impact in the effectiveness of parameter $l$ caused by impurities in the original graph, we again found that for $l = 2$, all precomputed data take up almost the same amount of space, indicating the content of the data is almost identical, with only slight difference in $k = 32$ and $k = 64$.

The growth of storage size by parameter $k$ is most like to be polynomial, as shown in Figure 14, where the data can be fitted against a degree 2 polynomial function. The slight error between the data and the fitted function can be explained by the inability of our partition function to create even-sized partitions, which causes the storage spaces needed for some partitions may be larger. The growth of storage size by parameter $l$ is clearly exponential, as shown to be fitted perfectly against an exponential function in Figure 15.

Since the difference between the growth of storage size by parameter $k$ and parameter $l$, we advise one to prioritize $k$ before $l$ when choosing a set of parameter for the preprocessing part of our algorithm.
3.4 Result Discussion and Suggestions

For the use of our algorithm inside as a component of the Incident Management System, we recommended using a setting of at least $k = 64$ and $l = 4$. Judging from the results of the experiments above, the settings of $k = 64$ and $l = 4$ is sufficient for the use in our preprocessed data of the Hong Kong road network. This would give our algorithm an average speedup of 13 when comparing with traditional Dijkstra’s Algorithm.

Given better data with the impurity removed, the settings of $k = 64$ and $l = 3$ will give a similar result as ours. By pushing the value of parameter $l$ to 4 although is it possible for a much better routing for the areas within the partitions, this probably would some but not as much to the performance of the algorithm as from Figure 11 and Figure 12, further divide a partition with $k = 64$ yields the size of sub-partitions as small as only one node. This mostly reaches $l_{\text{max}}$, however, indicating for $k = 64, l = 4$ would give the most out of our algorithm. It is, however, would need a much larger storage space since storage space increase exponentially when we adjust parameter $l$. For the use in Hong Kong, we would suggest an increase in the parameter $k$ instead, which the storage space needed would be smaller as the increase rate is only polynomial.

As for using our algorithm outside Hong Kong, we would suggest either using the same set of parameter as above if the target area has a comparable as Hong Kong, and using a larger $k$ for fine partitions and probably also larger $l$ for better details when routing at a local position, if the server can handles the size of preprocessed data.

The weakness that can be seen from the TsW-KC test routes using our algorithm is unfortunately unavoidable as the result of the TsW-KC test routes is most like caused by the road network but not the algorithm. This can be easily identified as the case as even the original Dijkstra’s Algorithm give a horrible result for the test routes. In order for us to overcome this weakness, we suggest removing these extremely slow roads that lead dead ends from the shortest path search, and instead direct the search to the entrance of that road segment instead.
4 Difficulties encountered

During the course of the project, we encounter a few problems and setbacks such that the progress of the project has been hindered. In this part of the report, a selection of such difficulties was highlighted and have been commented on.

4.1 Different iterations of algorithm designs

One of the largest setback during the course of the project is the different iterations of algorithm designs. Since shortest path routing problem in road networks is already well studied by others, it is difficult to design an algorithm that can provide a comparable speedup to other algorithms. This is further worsened by the fact that unlike shortest path routing, pass route awareness in shortest path routing is under-studied such that there is extremely little information that we can benefit from. This leads to a number of failed algorithm designs created during the first half of the project, including one of the well-elaborated designs that later proved to be flawed. Time spent was wasted during these failed iterations that can be used in the implementation and testing of the algorithms.

4.2 Implementation of the algorithm

There are a few difficulties concerning the implementation of the algorithm into the backend service. From the algorithm described earlier in this report, the confusing and complex nature of the precomputed information, as well as the information added/changed by programmatic optimizations, the data structure that we should use is difficult to determine. We finally resort to the C++ STLs as they are much flexible than primitive types. This, however, creates embarrassing scenarios where a complex compound container is used to convey specific information. This may increase storage size overhead for the precomputed data but it is difficult to tell. Since C++ is used in the implementation, there are limitations of C++ that causes some of the difficulties, one of which is memory management. Instead of auto memory management in modern languages such as garbage collector in JAVA, memory management in C++ is mostly manual. This causes issues in a system where the system is expected to have a long running time. Paring with the complex compound containers we decided to use, extra care is needed to prevent memory leaks from our implementation.
4.3 Testing of the algorithm

In order to test and perform experiments on our algorithm, we needed to prepare a large number of variations of precomputed data with different parameter settings. This poses a great challenge for us because the preprocessing process requires a long time and also much memory to do so. Because the delays of the algorithm design and thus the implementation of it, time was not enough for an extensive test on different parameters. The situation is much worsened by the fact that our system is designed to be a LINUX based system, and all except one of the team members do not have a native LINUX system and must resort to using virtual machines with LINUX installed. Although the LINUX operating system in virtual machines is capable of routing, there is not enough memory for preprocessing. It is also important to note that, the memory issues are raised in parts where external libraries are used, which we do not have control in and thus cannot reduce memory usage by any means. It is suggested for practical usage of the system that a more powerful machine, which is comparable to those used by other researchers[1][9][10][12], should be used for better performance for both preprocessing and routing.
5 Conclusion

In summary, we are given the fact that traffic management is an important issue for modern large cities, and Hong Kong is no exception. Since traffic management itself is a cross-discipline topic, most methods are beyond limits of what we can achieve. There is, however, one part of traffic management that we can work on, which is route suggestions for drivers that can reduce traveling time on the road network. Route suggestions is a widely explored and studied topic, that there exist a large number of works on finding the shortest route between points effectively. Most of these works, however, are not designed to work on extremely dynamic road network such as those in Hong Kong, and also do not handle the case that previously provided routes may have an impact on the overall road networks. This may at first do not seems to pose a large problem, but it may be the case when automatic driving become dominant.

Given this current situation, this project tries to create a routing service, IMS, for Hong Kong that not only it can provide route suggestions in a short period of time, the service should also be aware of the previous routed request, so that the newly routed path should also be the shortest path when part of the road network got congested or slowed by the previous routed request. This report highlighted one of the main parts of the whole system, the algorithm used for shortest path routing. Other parts such as previous path awareness, backend API, and frontend application were discussed in other reports written by the other team members.

This report firstly gives a brief summary on currently established speedup methods of short path search on road networks such as CH, PCD, and ALT, and show that these methods mostly expect a static road network and is not applicable in our system. Next, it introduces a new algorithm for IMS that was based on the A* algorithm while taking inspirations from both PCD and ALT. As with most other speedup methods, our algorithm also takes advantage of preprocessing the map graph into data that we can use during querying. The algorithm uses a layered partition approach throughout preprocessing and query that provides a better lower bound for each edge in the road network in accelerating shortest path search. Finally, the report describes a series of experiments and tests done on the implemented algorithm, the parameters of \( k = 64 \) and \( l = 4 \) were selected as recommended when using in the IMS. We identifying an average speedup of 13 when compared with the original Dijkstra’s Algorithm, the only comparable algorithm according to the restrictions given by this project.

Finally, this report also gives some suggestion for usage of this algorithm and the IMS in other places than Hong Kong by increasing the parameters of \( k \) more and \( l \) slightly. The report also discusses the case of slow local road segments that affect the overall performance of the algorithm, and the way the overcome it by some engineering compromises on the routing request itself.
References


