

國立交通大學

資訊科學與工程研究所

碩士論文

導航系統路徑搜尋之研究

On the Routing Problems of Navigation System

研究生：林志晏

指導教授：蔡錫鈞 教授

中華民國九十八年七月

導航系統路徑搜尋之研究

On the Routing Problems of Navigation System

研 究 生：林志晏

Student : Jhih-Yian Lin

指導教授：蔡錫鈞

Advisor : Shi-Chun Tsai

國立交通大學
資訊科學與工程研究所



Submitted to Institute of Computer Science and Engineering

College of Computer Science

National Chiao Tung University

in partial Fulfillment of the Requirements

for the Degree of

Master

in

Computer Science

July 2009

Hsinchu, Taiwan, Republic of China

中華民國九十八年七月

導 航 系 統 路 徑 搜 尋 之 研 究

學生：林志晏

指導教授：蔡錫鈞

國立交通大學資訊科學與工程研究所碩士班

摘 要

路徑規劃問題是這幾年來被廣泛地討論的一個問題，有許多基於 Dijkstra 演算法的方法已被一一提出；其中優先權佇列是在 Dijkstra 演算法的計算上非常重要的一個資料結構。我們針對一般平面道路圖來加以觀察，發現一般平面道路圖的節點分支度是相當地低。根據此特性我們提出一個 Lazy Heap 的資料結構。跟傳統常使用的 Binary Heap 來做比較，我們的方法能夠改進效能 30%至 50%。除此之外，我們針對台北大眾運輸系統，提出圖的模型和路徑規劃的系統架構；在基於不同使用者的偏好之下，我們設計不同評斷路徑好壞的方法。我們也嚐試許多不同的參數組合，來討論路徑規劃結果和運算效能上的改進的議題。

On the Routing Problems of Navigation System

student : Jhih-Yan Lin

Advisors : Dr. Shi-Chun Tsai

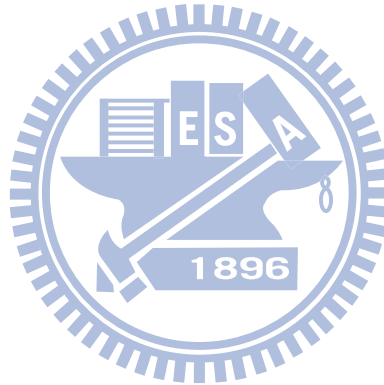
Institute of Computer Science and Engineering
College of Computer Science
National Chiao Tung University

ABSTRACT

The shortest path problem has been studied for decades, and many efficient algorithms based on Dijkstra's algorithm have been developed. In this thesis, we focus on the road network graphs. Priority queue is an important data structure used in Dijkstra's algorithm. Since the road network graph has low degree, we propose a lazy binary heap as the priority queue. According to our experiments, we reduce 30% to 50% running time on this kind of sparse graphs. Moreover, our implementation has better space efficiency than standard binary heap. Based on the road network, we also study the public transportation system of Taipei City. In this work, we first model it as a graph, then propose a system architecture. Because of the preference of different users, we design different metric functions for different preference. In our experiments, we have tried different metrics and parameters setting.

Acknowledgements

I am grateful to my advisor, Dr. Shi-Chun Tsai, for his guidance and encouragement. I also thank my family and friends for their spiritual support. Also, I am obliged to all members in CCIS lab for their help during my study. Specially, thank truck, Chun-Yen and tomos' help and discussion.



Contents

Abstract in Chinese	i
Abstract in English	ii
Acknowledgement	iii
Contents	iv
List of Figures	vi
List of Tables	viii
1 Introduction	1
2 Preliminaries	6
3 Efficient Priority Queue Implementation	11
3.1 Space Issue for Lazy Heap	13
3.2 Time Analysis	16
3.3 Experimental Results	18
4 Public Transportation Navigation System	22
4.1 Public Transportation System Model	23
4.2 Public Transportation Routing System	24
4.3 Public Transportation Routing Experiments	27
4.4 Experiments with Heuristic Functions	34



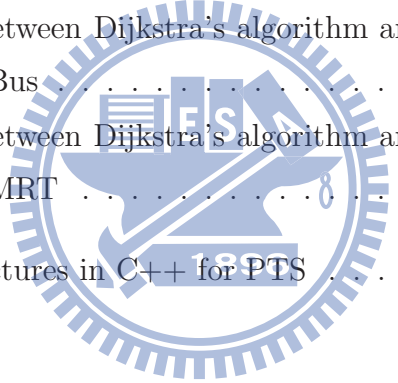
5	Conclusion	44
A	Some Implementation Issues for PTS	48



List of Figures

2.1	Dijkstra's algorithm	7
2.2	A* search algorithm	10
3.1	Modified A* search algorithm with lazy heap	12
3.2	An example shows that lazy heap cannot complete the route if update is not used for lazy heap.	15
3.3	The Implementations of Lazy Heap and Standard Binary Heap	20
4.1	The diagram for solving itinerary problem.	25
4.2	Algorithm for Path Finding	28
4.3	Path Extension	29
4.4	Algorithm for Path Extension	30
4.5	The result with walking distance limit as 400m.	31
4.6	The result with walking distance limit as 600m. This matches the result of Google Maps.	32
4.7	The second result with walking distance limit as 600m. This result is not found in the search of Google Maps.	33
4.8	Measure by (<i>transfer, price</i>) order.	35
4.9	Measure by (<i>transfer, time</i>) order.	36
4.10	The Speedup between Dijkstra's Algorithm and A* search with heuristic function Fare by applying preprocessing tech- nique of kd-tree	39

4.11	The Speedup between Dijkstra’s Algorithm and A* search with heuristic function Transfer by applying preprocessing technique of kd-tree	40
4.12	The Speedup between Dijkstra’s Algorithm and A* search with heuristic function Time	41
4.13	Comparison between Dijkstra’s algorithm and heuristic function Fare_Bus	41
4.14	Comparison between Dijkstra’s algorithm and heuristic function Fare_MRT	42
4.15	Comparison between Dijkstra’s algorithm and heuristic function Fare_Manhattan_MRT	42
4.16	Comparison between Dijkstra’s algorithm and heuristic function Transfer_Bus	43
4.17	Comparison between Dijkstra’s algorithm and heuristic function Transfer_MRT	43
A.1	The data structures in C++ for PTS	49



List of Tables

2.1	The Comparison among Different Priority Queue Implementations [21]	8
3.1	Experimental Data Set	19
3.2	The Experimental Results	21
4.1	A table for heuristic functions whose #paths are different from those of Dijkstra's. We test 1000 cases for each heuristic function.	40



Chapter 1

Introduction

The Shortest path problem (SPP for short) appears in many fields, network routing, transportation system, operations research, schedule planning and so on. Especially, since Global Position System (GPS) is open for everyone, there is a related industry whose business is to develop the service for finding the optimal route in a road network. There is one good example: a traveler probably uses his/her personal navigation device (PND) to find out how to reach destination as soon as possible in a city that he/she has never been. Therefore, SPP for optimal route becomes important in recent years. Although there are many algorithms for the shortest path problem, e.g. Dijkstra's algorithm [20], but the calculating time is still unacceptable for road network route application. For example, the number of vertices of a graph can be over 10 millions in the USA. If we apply Dijkstra's algorithm which costs $O(m + n \log n)$ time for n vertices and m edges, it might take too much time for route planning. Dijkstra's algorithm [20] is a classical algorithm for route planning and the size of the search space is $O(n)$. In order to speedup the performance, bi-directional search simultaneously performs forward and backward search. Moreover, the generalization of Dijkstra's algorithm, A^* search [19], is proposed and its bi-directional version [22] is also studied. While computing the shortest path, a priority queue is needed. RAM (Random Access Memory) model, is a abstract machine whose memory is broken

into W words each, and the machine accesses each memory word in one step. Based on RAM model, Mikkel Thorup implemented a queue supporting **find-min** operation in constant time, **delete-min** operation in $O(n \log \log n)$ [27]. Then he implemented a priority queue in linear space with n integer keys in the range $[0, N)$ supports **find-min**, **insert** and **dec-key** in constant time, and **delete** in $O(\log \log \min\{n, N\})$ [28]. From his work, the immediate result is that we can solve the single source shortest path problem (SSSP) for a directed graph G with n vertices and m edges with weights in a range $[0, C)$ in $O(m + n \log \log \min\{n, C\})$ time and in linear space.

People may plan their trips using various devices, such as PNDs, smart phones, laptops, desktops, and remote servers like Google Maps. We can classify them into two routing models: thin client model and off-line computing model. The major difference between thin client model and off-line computing model is web access. For thin client model, users can connect to a web service provider which supports online path query service, e.g. UrMap and Google Maps. Then user queries the path from a place to another from the service provider for road network routing and finally the route path information is shown on the web page. In this case, the computing power is from the remote server and the remote server is very powerful. When there are many users use the service at the same time, the calculating time for each query is a challenge.

The other model is off-line computing model. Many corporations, for example, Garmin, Mio, and TomTom, sell their personal navigation devices equipped with slower CPU (500MHz), limited fast memory space (64Mb) and larger slow storage (4Gb Flash memory or Disk). The road network graph data is stored in the slow storage of PND. When users have problems about the path information, they can use their PND to route the path and then PND will navigate to the destination. Because of slow computing power and limited memory space on this kind of small device, the system efficiency is a challenge. In order to achieve hardware optimization, the cache oblivious algorithm is introduced. Cache oblivious model is vary similar to RAM

model; in addition, cache is also introduced. The major difference between RAM and cache oblivious model is that a load or a store between main memory and a CPU register is serviced by cache. In [8, 9], they studied the characteristic of a cache and proposed the optimized priority queue. Since the graph representation cannot be loaded into main memory entirely, for example, the USA road networking data consists of over 10 million vertices, the data is divided into several blocks. These divided blocks are stored on the external storage initially. If some block is needed while routing the path, the block is then loaded into main memory. CPU's speed is very fast, but the speed of auxiliary memory is still slow. Because of a speed gap between CPU and the external storages, the I/O access is the bottleneck of computing. How to store the graph representation becomes one of critical issue recently, especially on small devices. There are many related works [5, 6, 7, 10, 11] based on discussing I/O complexity to handle SSSP and related problems.

To avoid reading map data for each query from slow secondary storage, developers might choose a method with small preprocessed data in order to load all map data in the memory. Laptops and desktop PCs are designed for one user with a powerful processor and plenty of storage. Developers might choose to minimize the data read from the hard disc for one query. Compared with the other platforms, PNDs and smart phones have slower processors, less memory and less space of secondary storage to keep higher portability and lower cost. A good implementation for these platforms requires not only time efficiency but also space efficiency since the spaces of memory and secondary storage are unable to hold large preprocessed data. Developers need to adjust parameters carefully to meet space and time requirements.

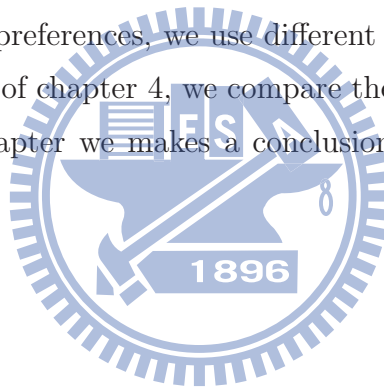
One approach to speedup the search is preprocessing. The main idea is that since the graph is static (this means the weight is unchanged), we can make some pre-computing and these results are attached to the original graph data. While routing a path, the additional information can help to reduce the search space; hence, the computing time is saved. Obviously, different preprocessing technique will cost different extra space and have

different speedup. This is a tradeoff. A naive way is to pre-compute all pairs of shortest paths and take $O(n^2)$ extra space, where n is the number of vertices. This may be practical for small graph; however, in real road network, we cope with a large graph which consists of at least one million vertices. This becomes impractical. The road network graph is planer and a lot of preprocessing techniques are based on dividing planer graph into hierarchical overlay graph. Highway hierarchy [18], Node Contraction [15] and Transit Node Routing [17] belong to this kind of technique. The Reach [16] and edge labeling [13, 12] techniques pre-compute the information of shortest path and then store it on edges or vertices. In order to take the advantages of these techniques, the hybrid technique is used in recent years, e.g. SHARC [14] hybrids hierarchy method and edge labeling technique.

An extension of road network is public transportation system routing. The new application focuses on travelers rather than vehicle drivers. Comparing to road networking routing, public transportation system or time table graph is more complicated because different issues are considered, including time table information. There is an example: a foreign traveler, Bob, arrives to Taipei City and he wants to visit the National Palace Museum. Since he does not have a driving license, a possible solution is to take public transportation system. He again can use personal navigation device for public transportation system to find out how to reach the National Palace Museum. In this model the itinerary planning constitutes a set of line transfers and walking information. Since different traveler will have different concerned criteria on routing, the public transportation system might answer different route paths that depend on the user's preference. In [24], Pyrga et. al. studied the time table information and modeled the time table system into time-depended and time-expanded graphs. The fare for public transportation system is studied in [25], Tan and Leong proved path depend shortest path (PDSP) is an NP-hard problem, but suffix-k PDSP is not NP-hard and proposed a solvable algorithm. To solve the itinerary problem for an urban public transport system, Konstantinos et al. [26] proposed the dynamic

programming based algorithm.

Our motivation is that since Global Position System is extensively used on path routing, there are many critical issues and applications under development. In this thesis, we study a critical data structure used for routing and the public transportation navigation system. The rest of this thesis is organized as follows. We first introduce the preliminaries and related work in chapter 2. In chapter 3, we study the data structure “lazy heap” and state its efficiency. Based on the same memory limitation, we describe a solution and experimental results. In chapter 4, we extend the road network to public transportation system. We also model the public transportation system of Taipei City and propose the architecture of public transportation system. Due to different user’s preferences, we use different heuristic measure methods. In the last section of chapter 4, we compare them with the results from Google. In the last chapter we makes a conclusion and discuss the future work.



Chapter 2

Preliminaries

In this chapter, we review some background for path routing. Consider the road network as a directed graph $G = (V, E)$, where $|V| = n$ vertices and $|E| = m$ edges. In general, $m = O(n^2)$ edges for the road network graph. Each edge $e = (u, v) \in E$ associated with a nonnegative cost $c(e)$. A path is a sequence of vertices (v_0, v_1, \dots, v_k) such that $(v_{i-1}, v_i) \in E$ for $1 \leq i \leq k$. The cost of a path (v_0, v_1, \dots, v_k) is defined as $\sum_{i=1}^k c(v_{i-1}, v_i)$. A path (v_0, v_1, \dots, v_k) is an s - t path if $v_0 = s$ and $v_k = t$. The shortest s - t path is the least cost one among all s - t paths. The point-to-point shortest path problem on graph $G = (V, E)$ with cost function c is to find the shortest s - t path from a given source $s \in V$ to a given target $t \in V$.

The classical algorithm for shortest path is Dijkstra's algorithm [20], see figure 2.1. Dijkstra's algorithm maintains three tables: tentative distance table $d[v]$, parent vertex table $p[v]$, and status table $S[v] \in \{\text{unreached}, \text{labeled}, \text{scanned}\}$ for each vertex v . Initially $d[v] = \infty$, $p[v] = \text{nil}$ and $S[v] = \text{unreached}$ for each unvisited vertex $v \in V \setminus \{s\}$; for the start vertex s , $d[s] = 0$, $p[s] = \text{nil}$ and $S[s] = \text{labeled}$. If there is any labeled vertex in S , select the labeled vertex u with minimum tentative cost. If a labeled vertex u is selected, $S[u] = \text{scanned}$ and then relaxes these edges which are out from vertex u . For each relaxed edge (u, w) , $S[w] = \text{labeled}$; update the $d[w]$ to $d[u] + c(u, w)$ and $p[w] = u$ if $d[w] > d[u] + c(u, w)$. Algorithm terminates

when there is no labeled vertex in S , and the tentative distance $d[v]$ is the shortest path from s to v . Dijkstra's algorithm computes all pair paths from vertex s ; however, our shortest path problem is only a query of shortest path from vertex s to vertex t . Therefore, we can modify the termination condition of Dijkstra's algorithm when $S[t]$ of the target vertex t is set as *scanned*.

Algorithm DIJKSTRA(G, c, s)

Input: $G = \langle V, E \rangle$ //the graph.

c //the cost function.

s //the source vertex.

Output: d //a table contains the cost of shortest paths where $d[t]$ is the least cost of s - t paths.

Variable: d //tentative distance table.

p //parent table.

S //status table.

begin

1. $d[s] := 0, p[s] := nil, S[s] := labeled;$
2. **for all** $v \in V - \{s\}$ **do** $p[v] := nil, d[v] := \infty, S[v] := unreached;$
3. **while** (there is a labeled vertex) **do**
4. $u :=$ the labeled vertex with minimum cost in S ;
5. $S[u] := scanned;$
6. **for all** $(u, v) \in E$ **do**
7. $d' := d[u] + c(u, v);$
8. **if** $(d' < d[v])$ **then**
9. $d[v] := d', p[v] := u, S[v] := labeled;$

end

Figure 2.1: Dijkstra's algorithm

To efficiently implement the algorithm, there are many data structures we can use. We additionally maintain a priority queue to collect these labeled vertices; we also call this priority queue as "open set". A priority queue is a collection of elements which are associated with priorities. The key of priority queue for shortest path algorithm is the tentative distance d . A priority queue supports following operations:

Extract-Minimum(Q): Returns an element of Q with minimum priority.

Insertion(Q,e): Adds the element e into Q .

Remove(Q,e): Delete the element e from Q .

Update(Q,e,k): Update the element e with priority value k .

In many application, for example, the shortest path problem, we only do the **update** operation if the priority value k is less than the value of element e in Q . In this case, the **update** operation is also called **decrease-key** operation. Different data structure used for open set will lead to different computing efficiency. Table 2.1 is the comparison of different data structure. Besides “open set”, we call the other data structure which is used to collect these scanned vertices as “closed set”. Because open set and closed set are applied, we can automatically classify the status of all vertices into three different sets and the status table S can be removed. If a vertex v in open set (closed set), the status of v represents *labeled (scanned)*. If a vertex v is not in open set and in closed set, the vertex is *unreached*.

	List	Binary Tree	Binary Heap	Fibonacci Heap
Insertion	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Extract_Min	$O(n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$ amortized
Update	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$ amortized
Delete	$O(n)$	$O(n)$	$O(\log n)$	$O(\log n)$ amortized

Table 2.1: The Comparison among Different Priority Queue Implementations [21]

The A^* search algorithm [19], see figure 2.2, is a generalization of Dijkstra’s algorithm for point-to-point shortest path. The main difference between Dijkstra’s algorithm and A^* search algorithm is the priority. The A^* search needs an additional heuristic function $h(v)$ to evaluate the priority of vertex v while Dijkstra’s algorithm only uses tentative distance $d(v)$, the cost of s - v shortest path. The heuristic function $h(v)$ is an estimation of the cost of v - t shortest path and the heuristic function is admissible only if it never over-estimates the actual cost. The optimality of the A^* search is

guaranteed with admissible heuristic function, otherwise the output of the A^* search algorithm might not be optimal. For computing path length between two locations in the real-world road network, the geometric distance is an admissible heuristic function. It is clear that Dijkstra's algorithm is the A^* search with heuristic function $h(v) = 0$ for all $v \in V$, and the search space decreases when the accuracy of heuristic function increases.



Algorithm $ASTAR(G,c,h,s,t)$

Input: $G = \langle V, E \rangle$, //the graph.

c //the cost function.

h //the heuristic function.

s //the source vertex.

t //the target vertex.

Output: opt //the least cost of $s-t$ paths.

Variable: Q //open set, an initially empty priority queue.

$closed$ //closed set, an empty set.

d //a table holds the cost of shortest paths.

p //a table holds the parent vertex for each vertex.

begin

1. $d[s] := 0, p[s] := nil$;

2. **for all** $v \in V \setminus \{s\}$ **do** $p[v] := nil, d[v] := \infty$;

3. Insert s with priority $h(s)$ into Q ;

4. **while** (Q is not empty) **do**

5. Extract minimum vertex in Q as u ;

6. **if** ($u \in closed$) **then continue**;

7. Add u to $closed$;

8. **if** ($u=t$) **then return** $opt := d[t]$;

9. **for all** $(u, v) \in E$ **do**

10. $d' := d[u] + c(u, v)$;

11. **if** ($d' < d[v]$) **then**

12. **if** ($v \notin Q$) **then**

13. Insert v with priority $d' + h(v)$ into Q ;

14. **else**

15. Update the priority of v in Q to $d' + h(v)$;

16. $d[v] := d', p[v] := u$;

end

Figure 2.2: A^* search algorithm

Chapter 3

Efficient Priority Queue Implementation

Different implementations of the priority queue used in Dijkstra’s algorithm lead to different time and space performance [28, 27, 9, 8, 23]. For example, it takes $O(n^2)$ time while using an array, but it only takes $O(n \log n)$ time with $\Omega(n)$ extra space while using a Fibonacci heap [23]. One might think that using binary heap as priority queue will not cause extra space usage because it is an in-space data structure. However, binary heaps are in-space only if they do not support any sub-linear time decrease-key operation, since finding an element without extra space needs $\Omega(n)$ time for an n -element binary heap. Typically, we use a hash map or a binary search tree to store the indices of element in the binary heap. Therefore, there is also $\Omega(n)$ extra space for standard binary heap implementation. In this thesis, we use binary heap as one of our experimental data structures. For standard binary heap, operations of extracted minimum, insertion, and remove all cost $O(\log n)$. In addition, we apply a hash map to record the stored position of an element in the heap; thus, the decrease-key operation has cost $O(\log n)$.

We propose a data structure called “Lazy Heap”. Our basic idea to improve the time efficiency is to use the binary heap in a lazy manner: avoid using the decrease-key operation. In the lazy heap implementation,

Algorithm MODIFIEDASTAR (G, c, s, t)

Input: $G = \langle V, E \rangle$ //the graph.

c //the cost function.

s //the source vertex.

t //the target vertex.

Output: opt //the least cost of s - t paths.

Variable: Q //open set, an initially empty priority queue.

$closed$ //closed set, an empty set.

d //a table holds the cost of shortest paths.

p //a table holds the parent vertex for each vertex.

begin

1. $d[s] := 0, p[v] := nil$;
 2. **for all** $v \in V \setminus \{s\}$ **do** $p[v] := nil, d[v] := \infty$;
 3. Insert s with priority $h(s)$ into Q ;
 4. **while** (Q is not empty) **do**
 5. Extract minimum element u from Q ;
 6. **if** ($u \in closed$) **then continue**;
 7. Add u to $closed$;
 8. **if** ($u=t$) **then return** $opt := d[t]$;
 9. **for all** $(u, v) \in E$ **do**
 10. $d' := d[u] + c(u, v)$;
 11. **if** ($d' < d[v]$) **then**
 12. Insert v with priority $d' + h(v)$ into Q ;
 13. $d[v] := d', p[v] := u$;
- end**

Figure 3.1: Modified A^* search algorithm with lazy heap

the decrease-key operation is not used. Therefore, lazy heap only supports insertion and extract minimum operation. Because of no decrease-key operation used, we do not need to check whether any vertex is in open set and thus the hash map or search tree is not used. Consequently, the running time for searching an element is saved. However, a problem occurs: a lot of insertions are needed. According to our observation, this modification brings the benefit on road network graphs. The reason is that the average degree of road network graphs is less than 4 and the extra insertion operations does not happen frequently. The algorithm ModifiedAStar, figure 3.1, is the modified A^* algorithm applies lazy heap as open set. We may have duplicated vertices in the open set when we apply lazy heap implementation. Thus, when a vertex u is extracted from open set, we should check whether u is already in closed set. If yes, this means the relaxation of vertex u is redundant and we can directly do the next extract minimum operation.

While inserting a vertex, there are two possible operations if standard binary heap used: decrease-key if the id of the vertex already exists and the priority is less; otherwise, insert it into open set. But lazy heap only does insertion operation no matter whether the vertex is already in open set. Since both methods relax the same vertices and extract the same vertex with minimum cost, the search scenario between standard binary heap and lazy heap are the same. However, lazy heap can speedup the search when the graph is sparse and in general road network is a sparse graph. The experimental results will be discussed later.

3.1 Space Issue for Lazy Heap

Now, we discuss a space issue between lazy heap and standard binary heap. As so far we known, because lazy heap only support insertion operation instead of using decrease-key operation, in general the lazy heap needs more space than that of standard binary heap (we can use more space because of the extra space for this case). In order to use the same space size as

the standard binary heap does, we propose a new approach. Let variable “*capacity*” be the maximum vertex size we can use while computing the shortest path. Every inserting operation event occurs, we first check whether the size used in heap is equivalent to *capacity*. If so, we do the following operations:

1. Sort the elements in heap by $(id, cost)$ lexicographical order.
2. Eliminate duplicated vertices.
3. Insert the new vertex into heap.
4. Do minimum heapify.

At setp 1, since there are many duplications with the same id, we can order the elements in heap by $(id, cost)$ lexicographical order. At step 2, we remove the redundant vertices; that is, for each vertex is in heap and is not in closed set, only keep one copy with minimum cost. After step 2, the heap structure may not satisfy the minimum heap property. Thus, we do the minimum heapify operation to achieve the requirement at step 3. Finally, we can insert the new vertex into heap if heap capacity is not full.

The above approach may cover many possible cases, but not all. We give a scenario that standard binary heap can complete the routing with limited space, but lazy heap cannot. Consider the graph of figure 3.2, and suppose for each edge $e \in E \setminus (0, 4)$, $c(e) = 1$ and $c(0, 4) = 3$. Let the *capacity* of our open set be 3 and suppose the SPP is a query from vertex 0 to vertex 3. Initially, vertex 0 is inserted into open set and then is extracted. After the relaxation of vertex 0, the vertices in heap are 1, 2, 4 and all are not in closed set. Now, suppose vertex 1 is extracted and relaxes its neighbors vertex 3 and vertex 4. If we apply standard binary heap, vertex 3 can be inserted into the heap and vertex 4 can be updated if the cost is less. Therefore, we can complete this relaxation and continue the route. Finally, the shortest path is found. However, if lazy heap is used, we cannot successfully route the path. After extracting vertex 1, and then relaxes vertex 3 and vertex 4 respectively,

vertex 3 can be successfully inserted into heap and the vertices in open set are 2, 3, 4. When vertex 4 is inserted, because the heap is full, we must do the above steps to eliminate these duplicated vertices. After completing elimination operation, the vertices in open set are still 2, 3, 4. The heap size is still full; therefore the insertion operation fails. In above case, we know the capacity needed for lazy heap is more than that for standard binary heap. However, if the decrease key operation is allowed to use when the above situation occurs, the space used can be reduced.

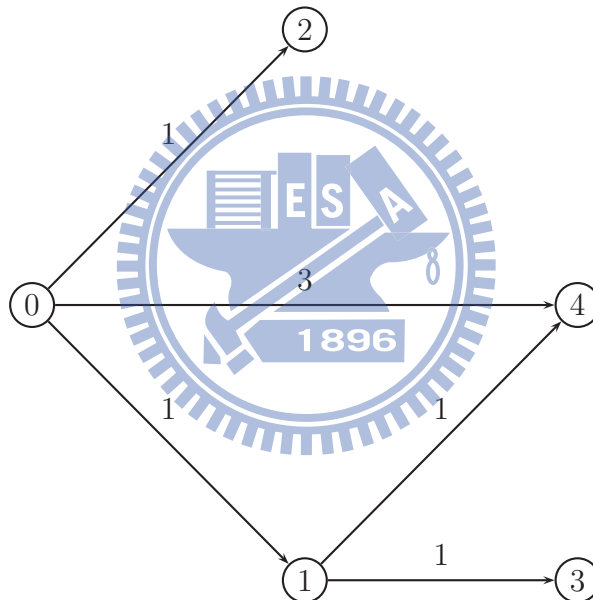


Figure 3.2: An example shows that lazy heap cannot complete the route if update is not used for lazy heap.

We must modify the algorithm as follows:

1. Sort the elements in heap by $(id, cost)$ lexicographical order.
2. Eliminate duplicated vertices.
3. If still full, do the update operation for the vertex and build the heap structure.

4. Else, insert vertex into heap.
5. Do minimum heapify for these vertices.

When we insert vertex 4, the vertex is updated at step 3. Therefore, the route path is also completed. We claim the above algorithm can complete the relaxation when it is also completed for standard binary heap. Since the update operation is used for standard binary heap, there is only one copy for each labeled vertices in it and there is at least one copy for these labeled vertices in lazy heap. After step 1 and step 2, there are only one copy for each labeled vertices with minimum cost in lazy heap and these scanned vertices will be removed from it. These elements in standard binary heap will also have one copy in lazy heap, and vice versa. At step 3, if the capacity size of lazy heap is still full, then the standard binary heap is also full. Suppose we can complete relaxation for the vertex in standard binary heap, this means the operation must be updated; otherwise, this vertex cannot be relaxed by insertion in a full standard heap. Now the update operation is allowed for lazy heap, the capacity of lazy heap will not be over-sized. Thus, we can complete this relaxation. Finally, we do the minimum heapify to meet the requirement. After these operations, the structure of two heaps are the same.

3.2 Time Analysis

In this section, we discuss the time efficiency of the modified priority queue (lazy heap). Since we are coping with a sparse graph with limited memory resources, we assume the following:

- The maximum degree of the graph is a constant d .
- The size of memory allocated for implementing heap is s .
- The capacities of the standard binary heap and the modified one are n and m copies, respectively.

- While executing the algorithm, the number of distinct nodes in the priority queue is no more than n .

Now we give some analysis of the modified priority queue. Since linear space data structure such as search trees and hash tables are usually used for the standard binary heap implementation to support sub-linear time decrease key operation, the lazy heap can hold more copies of node. Therefore, we assume $m = cn$ where $c > 1$ is a constant.

In the lazy heap implementation, there are only two operations insertion and deletion (extract minimum operation). The running time of extract minimum operation is clearly $O(\log m) = O(\log n)$. When the priority queue is full, removing unnecessary copies takes $O(m \log m)$ time. But there are at least $m - n = (c - 1)n = \Theta(m)$ insertion operations invoked between any two insertions making the priority queue full. Hence the modified insertion has an amortized running time $O(\log m) + O(m \log m) / \Theta(m) = O(\log n)$ under this setting. This analysis actually works for any constant $c > 1$. This allows us to adjust the actual space usage to fit the graph input and system resources. Hence the implementation have a space efficiency almost as efficient as an array, and a time efficiency near using a standard binary heap with a sub-linear time decrease-key operation.

We continue to analyze the time efficiency of the whole algorithm. The modification does not change the search space, therefore the vertices which have been inserted into the priority queue are the same. Consider an inserted vertex v in the standard version, assume that v has been inserted only once, extracted at most once and decreased d_v times. Since we replace the decrease-key operation with insertion, v is inserted $1 + d_v$ times and extracted at most $1 + d_v$ times in the modified version. Note that the amortized costs of these operations are all $O(\log k)$ where k is the number of copies in the priority queue, and there is at most d copies. Since the time complexity of heap operations of both versions are the same, the running time of the whole modified algorithm has the same order as that of standard version.

3.3 Experimental Results

We implemented the algorithms by using C++ language and GNU C++ STL. These programs were compiled by the GNU compiler 4.2.3. Our experimental platform is a machine equipped Intel Core2 1.86GHz CPU, system RAM 2G on an Ubuntu Linux distribution with kernel version 2.6.24. Our test instances are from [1] and the random test cases are planner graphs which are generated from LEDA [2]. For the DIMACS test instances, the average degree of the graphs are less than 3 and the average degree of the random planner graphs are larger than 5. Our classes declaration of lazy heap and standard binary heap are listed in figure 3.3. While inserting an vertex into a fully heap, the function *removeRedundant* of class *LazyHeap* removes the redundant vertices. The function pointer *cmp* points to a function for measuring a vertex priority in the heap. We apply the hash map for mapping stored position of labeled vertices in standard binary heap. We set the heuristic function $h(v) = 0$. This means all shortest paths are computed by Dijkstra's algorithm. For the detail information of test instances and experiment results please refer to table 3.1 and table 3.2, respectively.

Compare with the standard binary heap implementation, the experimental result shows that Lazy Heap can speedup by 30% and 50% on DIMACS test instances and random graphs, respectively. For the case of space consideration, the bounded lazy heap still has better performance than that of standard binary heap implementation. In general, the average degree of road network graph model is less than 4; thus, we believe routing data structure can apply lazy heap implementation as open set.

Since the road network becomes lager, the computing time of Dijkstra's algorithm increases significantly. The preprocessing technique is more and more important. An important application of lazy heap is in preprocessing [12, 13, 15, 16]. If the lazy heap is used, the preprocessing time may be reduced for different preprocessing methods. For example, the preprocessing technique arc flag needs to compute the shortest path and stores the shortest

path information on edges. From our experiments, the computing time is saved if lazy heap is applied. This is very useful to update the graph data online. According to our experiments for limited size lazy heap and standard binary heap, the execution time of our implementation is better. When remote server needs many simultaneously queries, we can save the query time without any extra space if use our implementation.

Test Case	Description	#Vertices	#Edges
NY	New York City	264,346	733,846
BAY	San Francisco Bay Area	321,270	800,172
COL	Colorado	435,666	1,057,066
FLA	Florida	1,070,376	2,712,798
NW	Northwest USA	1,207,945	2,840,208
NE	Northeast USA	1,524,453	3,897,636
CAL	California and Nevada	1,890,815	4,657,742
Random_1	Random Planer Graph 1	1,000,000	>5,000,000
Random_2	Random Planer Graph 2	1,000,000	>5,000,000

Table 3.1: Experimental Data Set

```

class LazyHeap{
private:
    unsigned int size;
    unsigned int capacity;
    Elem** array;
public:
    LazyHeap(int (*c)(Elem*, Elem*));
    ~LazyHeap();
    void removeRedundant();
    Elem* extractMin();
    void insert(Elem* e);
    int (*comp)(Elem* e1, Elem* e2);
    bool isEmpty();
};

class StdHeap{
private:
    unsigned int size;
    unsigned int capacity;
    Elem** array;
    hash_map<unsigned int, unsigned int> heapMap;
public:
    StdHeap(int (*c)(Elem*, Elem*));
    ~StdHeap();
    void decreaseKey(unsigned int id, double cost);
    Elem* extractMin();
    void insert(Elem* e);
    int (*comp)(Elem* e1, Elem* e2);
    bool isEmpty();
    bool find(unsigned int id);
};

```



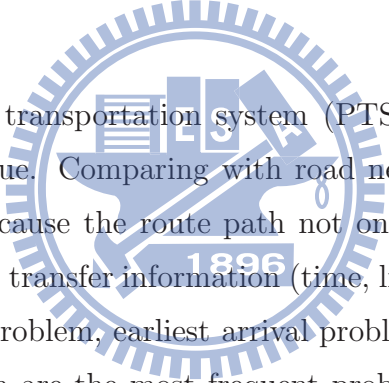
Figure 3.3: The Implementations of Lazy Heap and Standard Binary Heap

Test Case	Method	#seconds per 1000 Queries	Speedup
NY	Heap with decrease-key	476	-
	Bounded Lazy Heap	334	1.42
	Lazy Heap	325	1.46
BAY	Heap with decrease-key	542	-
	Bounded Lazy Heap	354	1.53
	Lazy Heap	353	1.53
COL	Heap with decrease-key	751	-
	Bounded Lazy Heap	490	1.53
	Lazy Heap	488	1.53
FLA	Heap with decrease-key	1989	-
	Bounded Lazy Heap	1316	1.51
	Lazy Heap	1314	1.51
NW	Heap with decrease-key	2203	-
	Bounded Lazy Heap	1414	1.55
	Lazy Heap	1410	1.56
NE	Heap with Update	2940	-
	Bounded Lazy Heap	1939	1.51
	Lazy Heap	1939	1.51
CAL	Heap with decrease-key	3620	-
	Bounded Lazy Heap	2382	1.51
	Lazy Heap	2377	1.52
Random_1	Heap with decrease-key	5571	-
	Lazy Heap	4040	1.37
Random_2	Heap with decrease-key	5760	-
	Lazy Heap	4123	1.39

Table 3.2: The Experimental Results

Chapter 4

Public Transportation Navigation System



In recent years, public transportation system (PTS for short) routing becomes an important issue. Comparing with road networking routing, PTS is more complicated because the route path not only depends on the road information but also the transfer information (time, line and so on..). In PTS routing, the least fare problem, earliest arrival problem and minimum number of transfers problem are the most frequent problems encountered. The least fare problem is a query of the connection departure from a location to another with least fare. In the earliest arrival problem, our goal is to search the least time used from a to b . There are two variations: constant transfer time and dynamic transfer time. In order to achieve the earliest arrival goal, we may need transfer many times. However, some users may prefer fewer transfers. Comparing with earliest arrival problem, the minimum number of transfers problem requires the minimum number of transfers but may take more time. In this chapter, we describe the PTS model for Taipei city and propose some approaches for routing.

4.1 Public Transportation System Model

In this section, we first model the Taipei City public transportation system as a graph. Let S be the set of stops of the public transportation system and N be the set of nodes. Each node v represents a service line that passes a specific stop s . A stop consists of the position at the map, its stop name and different service lines going through. We define the function $Stop : N \rightarrow S$ to map the corresponding stop information of a node. A service line can be represented as a sequence of nodes (n_1, n_2, \dots, n_k) . In general, the transfer time for taking a service line from one node to its next node is dynamic (depends on the level of traffic congestion). In this thesis, we simplify the transfer time in minutes of its travel distance in meter divided by the average speed of this transportation service. In Taipei City, there are two major public transportation systems: bus and Mass Rapid Transit (MRT). The average speed of bus and MRT are 15 km and 35 km per hour, respectively. We denote E the set of transfer edges. For each $e = (n, n') \in E$, n and $n' \in N$, let $transCost(e)$ be the cost function. Depending on user's setting, the transfer cost function will calculate the fare, the number of transfer and transfer time. In addition, we define the walk time $w(s_1, s_2)$ to represent the walking time from stop s_1 to stop s_2 . Let W be the set of walking edges. In our setting, the walking time in minutes is the travel distance divided by the walking speed (6km/h). The parameter *walking distance limit*, LW , is the maximum walking distance that a traveler allows for each service line transfer. The public transportation system can be modeled as a network, $G(S, N, E, W)$. Since different travelers may care about different issues, for example, a business traveler cares more about travel time but a student may care more about the travel price. We additionally define the measure function M for measuring two paths. An itinerary planning can be represented by a sequence $(l_s, s_1, \dots, s_k, l_t)$, where l_s and l_t represent the start location and target location, respectively.

4.2 Public Transportation Routing System

Figure 4.1 is our system architecture for public transportation routing. Initially, user can specify the parameters: start location, target location, maximum walking distance per transfer. The function *main* is a control flow function. We introduce the details about the interaction between the *main* function and other modules as follows.

The first module invoked is data reading module. This module has the functionalities: processing data and loading data. We first parse the raw data and write it into binary files. When *main* function invokes load map module, the binary files are directly loaded into corresponding data structures. The detailed preprocessing technique refers to [29]. Neighbor finding module will respectively return the set of neighbor stops when walking distance set by user of start location and that of target location. For the set of neighbor stops of start location, we can consider these stops as the candidates of starting transfer positions. User can walk to one of these stops from the start location, then taking the transportation lines going through these stops towards destination. Similarly, for the neighbor stops of target location, a user arrives to one of them; then walks to the destination within a distance which is set by the user. In order to efficiently search the neighbor stops, range tree [4] may be a choice. It takes $O(\log n)$ search time, but use extra $O(n \log n)$ space. Due to the space issue, we do not apply this method. In our method, we first sort the stops information by (longitude, latitude) lexicographical order while processing the transportation data. When neighbor finding module is invoked with parameters: geometric location and walking distance limit, it can calculate the ranges which we may need to take into consideration. In this implementation, it may take linear time to collect these stops. The implementation without extra space for graph data is acceptable because we only need to do neighbor finding two times (for start location and target location). The route paths are generated by the path computing module which applies Dijkstra's algorithm or A^* search. While each time we

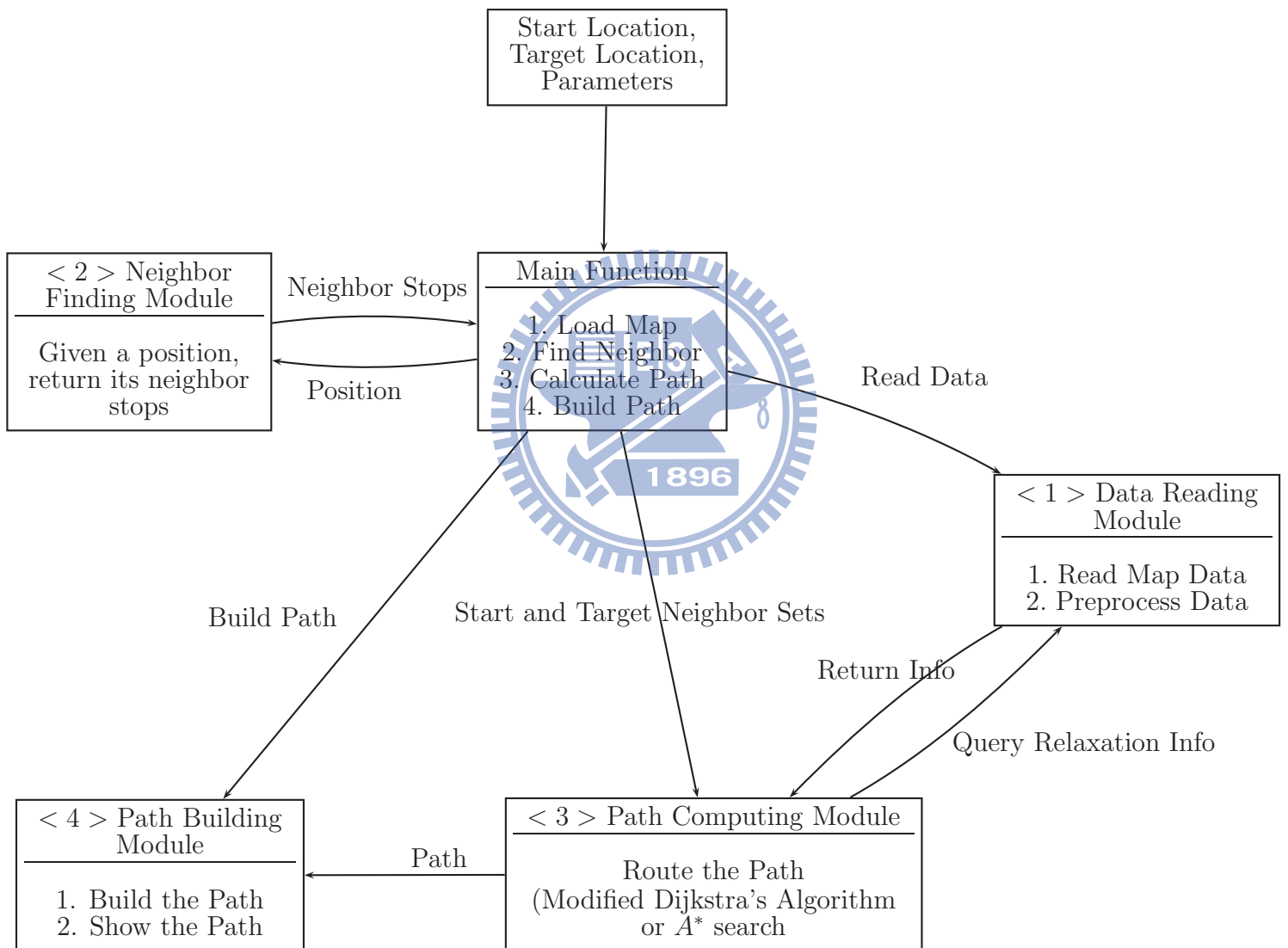


Figure 4.1: The diagram for solving itinerary problem.

arrive a stop and getting off is allowed, there is a possibility for changing lines toward the destination with less cost. Thus, while arriving to a stop each time, we must check the stop candidates that we can walk to. One solution is that we apply the “on the fly” search, which again applies neighbor finding module. For this method, there is no extra space needed if use our neighbor search method. However, this may spend more time on search. In order to reduce the search time, a possible solution is use the range tree implementation. Time complexity can be reduced to sub-linear time for each search. However, the search is frequent; the performance is still not good enough. The other method, we can pre-calculate the neighbors of each stop which are within a range of distance, then store on a set of lists. For each stop s , it has a neighbor list which represents the neighbor stops near s in a range. For each s , we sort its neighbor list by the distance from s . This method may take $O(n^2)$ extra space, but we do not need to search its neighbor when arrive to a stop each time. When the corresponding stop of extracted node belongs to the target set, there is a path we can go from start location to destination. This path will be passed to the path building module. Finally, the main function invokes the path building module to build the route path.

Figure 4.2 is the path finding algorithm. The input is the modeled graph G , parameters and functions. At first step, we calculate the cost for each node n whose corresponding stop belongs to the neighbor set $NeigS$. Dijkstra’s algorithm is applied from line 4 to line 20. At line 5, we measure the priority of nodes in open set by the specific metric function M , then extract the the minimum weight node from the open set. From line 6 to line 10, we check whether the corresponding stop of this extracted node belongs to target set $NiegT$. If yes, this means that we already arrive to a stop whose distance from it to the destination is less than the preferred walking distance. Therefore, we can build this path and then report the transfer information from the source location to the destination. Before building the route path, we first check whether there is a better stop which is much closer to destination at line 7. Our main idea is that since we arrive to a stop which

belongs to target set $NeigT$, traveler may need to walk to destination. In general, we must avoid walking a long distance. Therefore, we should check if there is any better stop which is closer to the destination. In figure 4.3, suppose a traveler arrives to stop 1 by taking line A. The distance between stop 1 and destination is less than WL (we suppose that these stops enclosed in a rectangle are all in $NeigT$). In this case, we may walk to destination in 500m; however, there is a better stop, stop 2, which is closer to destination(only 200m). Usually, the speed of walking is very slow(6km/h for our setting) and the speed of transportation tool is much faster. Although there is a better stop to get off, we need to check some property of this stop. We must ensure that the price charged to this stop is unchanged and this stop allows travelers to get off. Figure 4.4 is the algorithm to get the best stop. If the corresponding stop of the extracted node is not in target set, first we can directly relax its next point at line 11. This means we continue taking the same line. If we can get off at current stop, according to the above description, we can consider the neighbor stops and relax them from line 12 to line 20. The algorithm continue the search until a path is found or priority queue is empty (i.e., there is no path).

4.3 Public Transportation Routing Experiments

In this section, we experiment with different measure functions, walking distance limit for PTS of Taipei City. Because of the incompleteness of some information, we make up some data for the incomplete information. For each test case, the results are compared with those of Google Maps. We implemented the algorithms with C++ language and GNU C++ STL. Those programs were compiled by the GNU compiler 4.2.4. Our experimental platform is a machine equipped with Intel Core 2 Duo 2.20GHz CPU and system RAM 2G on an Ubuntu Linux distribution with kernel version 2.6.24. In order to easily observe the route path, we also build a web service version for experiments. The web service is builded by using Google Maps API, PHP,

Path Finding Algorithm

Input: $G\langle S, N, E, W \rangle$ //the graph where

S is the set of stops, **N** is the set of nodes,

E is the set of edges and **W** is the set of walking edges.

transCost //the cost function.

w //the walking time function.

s //the source location.

t //the target location.

LW //the walking distance limit.

M //the measure function.

Stop //mapping function from N to S .

NeigS //the neighbor stops of source location.

NeigT //the neighbor stops of target location.

Output: The s - t path which consists of the line information.

```
1: for each  $st \in NeigS$  do
2:   Calculate the cost of the corresponding node  $p \in N$  which belongs to
   stop  $st$ , then insert it into open set  $Q$ .
3: end for
4: while  $Q$  is not empty do
5:   Extract minimum node  $u$  from  $Q$  by using measure function  $M$ .
6:    $cs := Stop(u)$ 
7:   if  $cs \in NeigT$  then
8:      $curLine :=$  current taking line.
9:      $Path\_Extension(cs, curLine, NeigT)$ .
10:    Build the path from  $s$  to  $t$  with transfer information.
11:    return  $s$ - $t$  path.
12:   end if
13:   Relax the next node  $v$  that is reachable by taking the line of current
   extracted node  $u$  without any transfer.
14:   if we can get off at  $cs$  then
15:     Find the neighbor stops of  $cs$ , denote  $NeigCS$ .
16:     for each  $st \in NeigCS$  do
17:       if we can get on at  $st$  then
18:         Relax node  $p$ , such that  $Stop(p) = st$ .
19:         Update the cost of  $p$  and insert  $p$  into  $Q$ .
20:       end if
21:     end for
22:   end if
23: end while
```

Figure 4.2: Algorithm for Path Finding

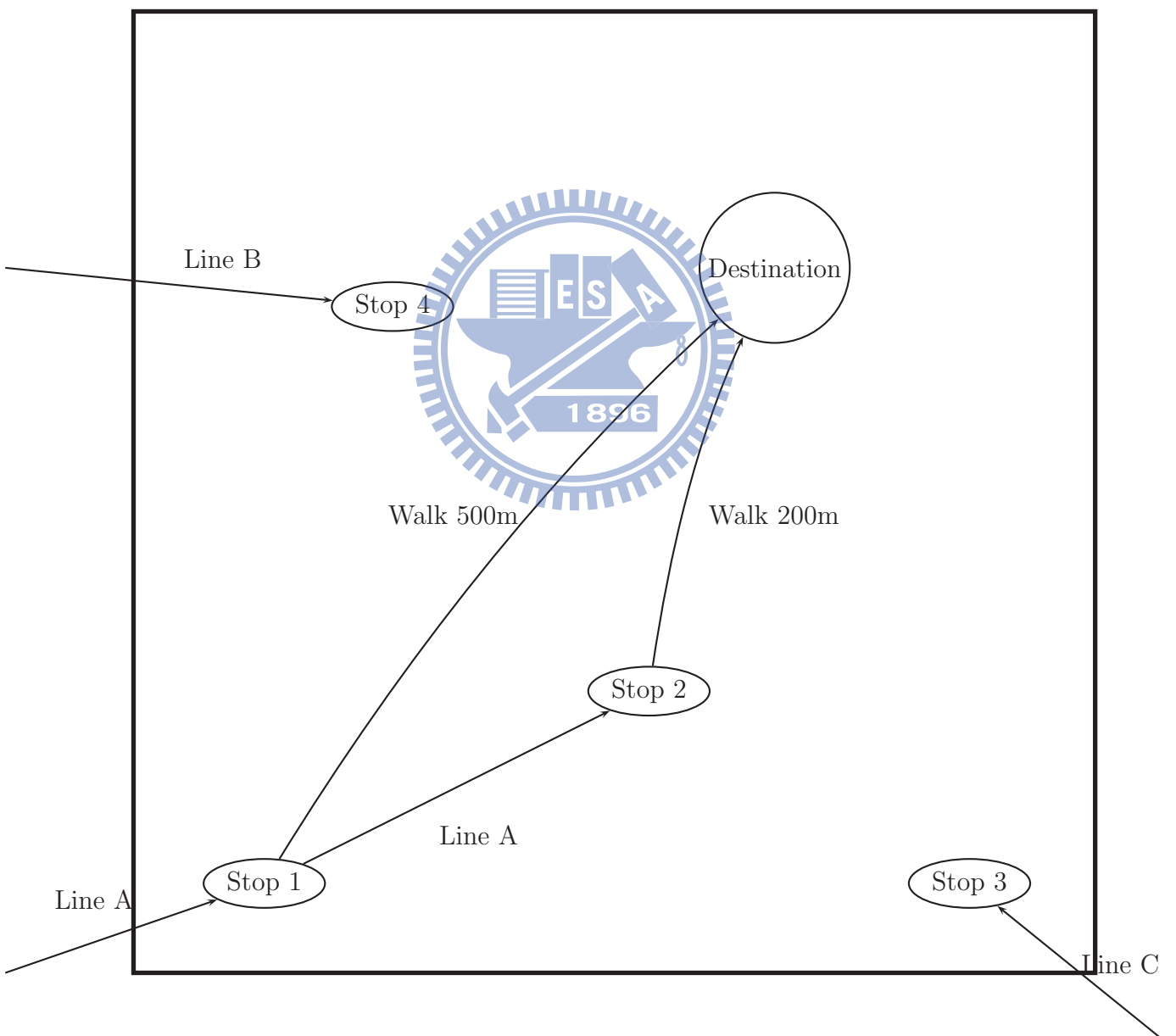


Figure 4.3: Path Extension

Path Extension Algorithm

Input: `curStop` //the current stop we arrive.

`curLine` //current line we are taking.

`NeigT` //the neighbor stops of target location.

Output: `best` //the best stop we will get off.

```
1: bestStop := curStop
2: for each st ∈ NeigT \ s do
3:   if curLine pass to st then
4:     if st is closer to destination and the fare is unchanged and we can
       get off at st then
5:       bestStop := st
6:     end if
7:   end if
8: end for
9: return bestStop
```

Figure 4.4: Algorithm for Path Extension

JQuery and the detail implementation we refer to [29].

First, we show the different paths caused by different walking distance limit setting. In figure 4.5, we query the path from Sanchong city to Songshan District, Taipei. If we set the walking distance as 400m, measuring by $(transfer, time)$ lexicographical order, the route paths need at least one transfer. However, Google Maps finds the two similar paths (taking the same bus 306) without any transfer. Now we relax the walking distance to 600m and the measure function is unchanged. In this experiment, we have three paths without any transfer: not only the bus 306 which is also the path found by Google Maps, but also a different path, which takes bus 622 to destination; see figure 4.6 and figure 4.7. In this experiment, we observe the importance of walking distance limit setting. Google Maps does not support the parameter of walking distance limit; therefore, while routing the path, Google Maps can consider more paths.

In the other case, we test the path from Banciao City to Da-an District, Taipei. We apply two different metrics, $(transfer, time)$ lexicographical or-

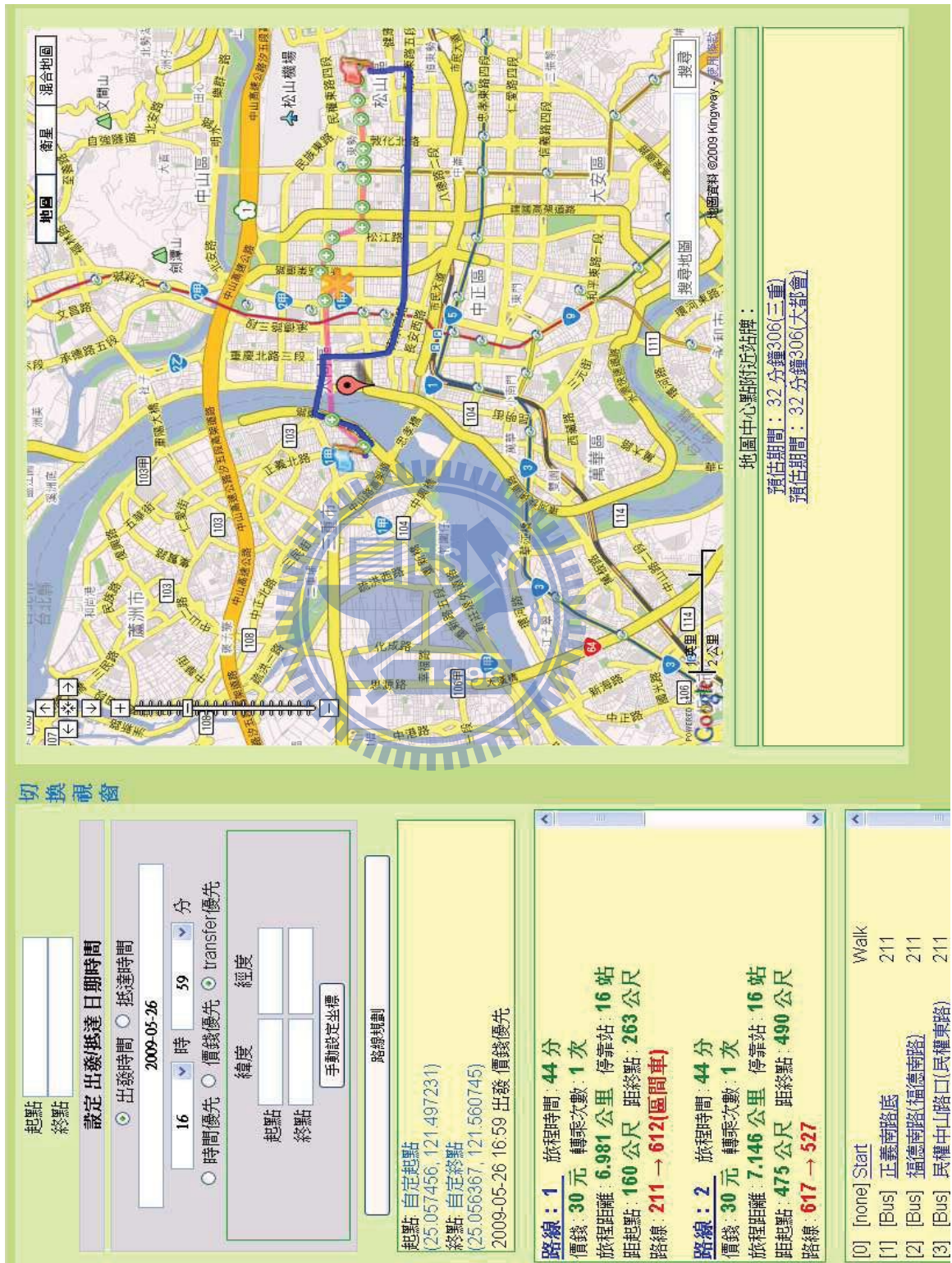


Figure 4.5: The result with walking distance limit as 400m.

起點

終點

設定出發抵達日期時間

出發時間 抵達時間

2009-05-26

16 時 55 分

時間優先 價錢優先 transfer優先

緯度

經度

起點: 自定起點
(25.057456, 121.497231)

終點: 自定終點
(25.056367, 121.560745)

2009-05-26 16:55 出發 價錢優先

地圖中心點附近站牌：

預估期間：32 分鐘(三重)

預估期間：32 分鐘306(大都會)

路線：1 旅程時間: 40 分
 價錢: 15 元 轉乘次數: 0 次
 旅程距離: 7.409 公里 停靠站: 16 站
 距起點: 160 公尺 距終點: 627 公尺
 路線: 306

路線：2 旅程時間: 41 分
 價錢: 30 元 轉乘次數: 1 次
 旅程距離: 7.566 公里 停靠站: 17 站
 距起點: 160 公尺 距終點: 627 公尺
 路線: 306 → 605(快速公車)

[0] [none] Start Walk 306

[1] [Bus] 正義南路底 306

[2] [Bus] 福德南路(福德南路) 306

[3] [Bus] 漳州重慶路口 306

Figure 4.6: The result with walking distance limit as 600m. This matches the result of Google Maps.

起點
終點

設定出發/抵達日期時間

出發時間 抵達時間

2009-05-26

16 時 55 分

時間優先 價錢優先 transfer優先

緯度 經度

起點
終點

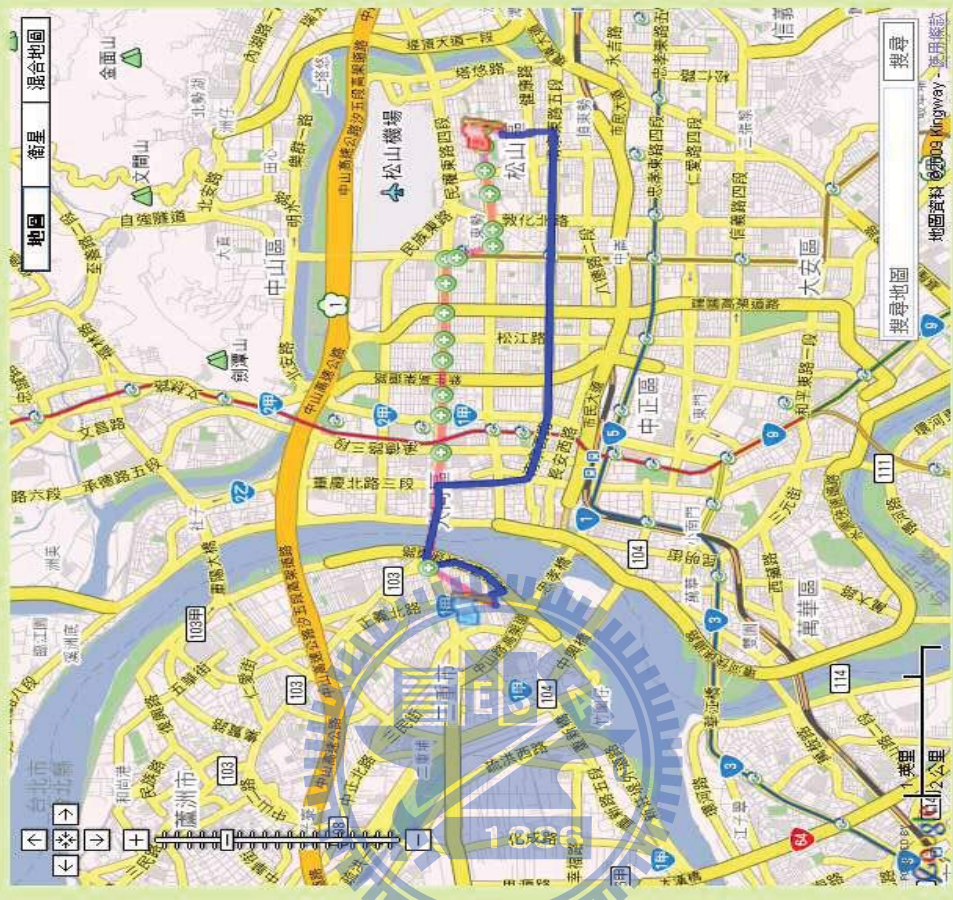
手動設定坐標

路線規劃

起點 自定起點
(25.057456, 121.497231)

終點 自定終點
(25.056367, 121.560745)

2009-05-26 16:55 出發 價錢優先



地圖中心點附近站牌：

預估期間：32分鐘306(三重)

預估期間：32分鐘306(大都會)

價錢：15元 轉乘次數：0次

旅程距離：7.56公里 停靠站：15站

距起點：551公尺 距終點：627公尺

路線：622

路線：5 旅程時間：50分

價錢：15元 轉乘次數：0次

旅程距離：6.318公里 停靠站：16站

距起點：897公尺 距終點：263公尺

路線：225

Figure 4.7: The second result with walking distance limit as 600m. This result is not found in the search of Google Maps.

der and $(transfer, price)$ lexicographical order. The walking distance limit is 500m. Figure 4.8 is a list of the route paths by using $(transfer, price)$ order. Since the fare is asked as cheap as possible, these paths all consider taking bus with one transfer (in general, the fare for MRT is more expensive). However, this might not match the results generated by Google Maps. In Figure 4.9, we can have two paths which match the paths generating by Google Maps, both take MRT first, then have a transfer to bus. For some other paths that we do not match that of Google Maps, we explain the possible reasons:

1. Our PTS data and that of Google Maps are different. Especially, a lot of our bus information is made up, which may not match the data of Google Maps. In most of the cases that our results are different from Google Maps' by taking different buses line.
2. The measure methods between ours and Google Maps' are not the same. This will cause different route results.
3. Different walking distance limit leads to different paths. In figure 4.9, the last paths is to take bus line 245 without any transfer; however, according to our observation, this path suggests traveler to walk over 1 km. For our setting of this experiment, we only allow walking at most 500m for each transfer. Therefore, bus line 245 is ignored by our routing algorithm.

4.4 Experiments with Heuristic Functions

Now, we discuss different metric methods concerned with travelers. There are many possible criteria that we can discuss, e.g. the number of stops we pass. However, the major issue is fare, travel time, walking distance and the number of transfer times. For walking distance criterion, we already have a parameter LW for this criterion. Based on these criteria, we generate different measure functions in lexicographical order. We design many lexicographical order functions for comparison. For lexicographical order, we focus

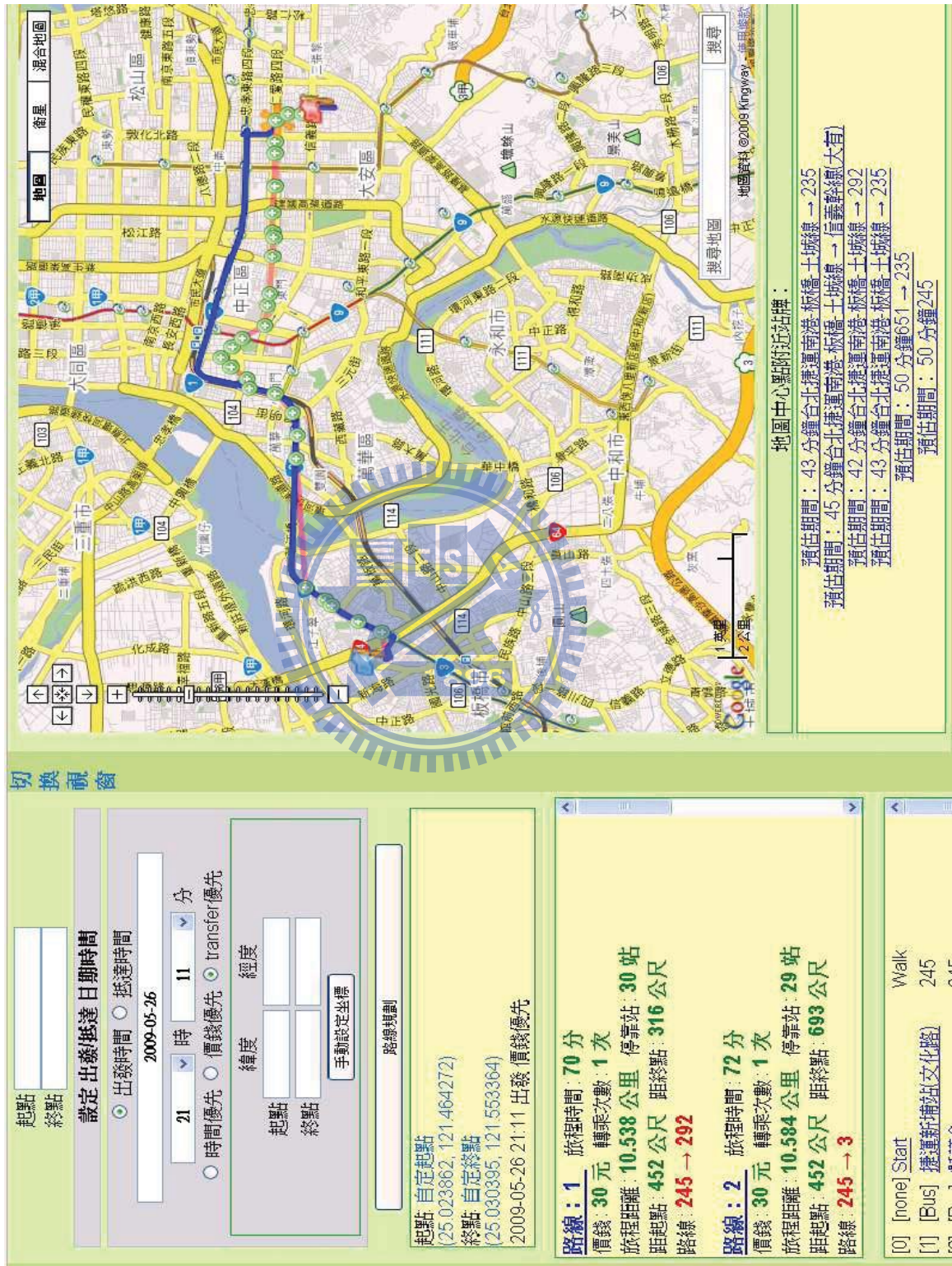


Figure 4.8: Measure by (transfer, price) order.

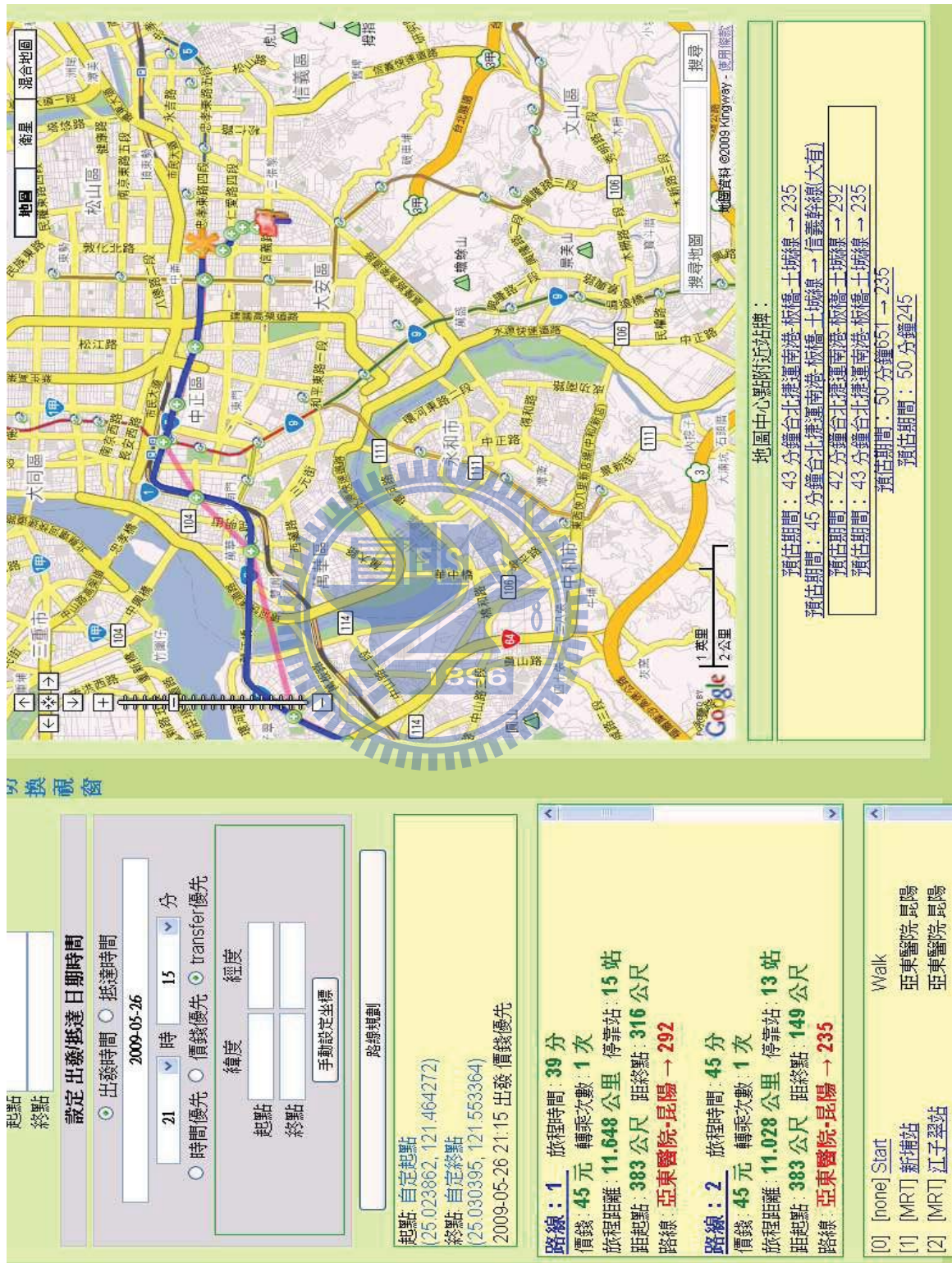


Figure 4.9: Measure by (transfer, time) order.

on three major criteria: travel time, the number of transfer and fare. We try several combinations.

Different heuristic functions setting may result in different route paths. The optimality of the A^* search is guaranteed with an admissible heuristic function, otherwise the output of the A^* search algorithm might not be optimal. Therefore, we must decide the admissible heuristic functions for travel time, fare and the number of transfers. For the travel time criterion, it depends on the transfer time, walking time and waiting time. In the optimal case, we can directly take this transportation carrier to the target location. In this case, no walking time and waiting time are needed. This leaves the transfer time undecided. In computing path between two locations in the real-world road network, the geometric distance is an admissible heuristic function. Our idea is that assume taking the highest speed transportation carrier to the target location (denote HS); therefore the value of geometric distance divided by HS is also under-estimated. For fare criterion, because the fare for bus depends on not only geometric distance; thus, the heuristic function for fare is much complicated. In order to complete the heuristic function, we apply the idea of preprocessing. Our idea is that first divide stops information into several regions, then preprocess the lowest fare from one region to other region. After completing preprocessing for fare, we record the data as a table. While query the fare from one stop to target location, according to the table, we can have the cheapest price to the target location, which is an under-estimation. In order to efficiently retrieve the corresponding region of a stop, we use kd -tree [3] to divide the regions. The time complexity to get the region of a stop is $O(l)$, where l is the level of the kd -tree. For the number of transfer issue, we also apply the preprocessing technique of kd -tree to pre-compute the minimum number of transfers from each region to other regions. Because of some engineering issue, we may not always need the optimal path. Thus, we can try some over-estimating heuristic functions. We additionally design over-estimating heuristic functions for fare and transfer criteria. For these heuristic functions, first we calculate

the distance between current position and target position by $\min(\Delta x, \Delta y)$ distance or Manhattan method. For fare criterion, we calculate the fare by MRT price or bus price. For the transfer issue we set parameters $\alpha_{bus}=25$ and $\alpha_{MRT}=15$. Besides, denote $speed_{MRT}$ and $speed_{bus}$ the speed of MRT and bus, respectively. The heuristic transfer cost for MRT is the value of $distance/(speed_{MRT} * \alpha_{MRT})$. Similarly, the heuristic transfer cost for bus is the value of $distance/(\alpha_{bus} * speed_{bus})$.

Finally, we experiment with the heuristic functions for A^* search. For each heuristic function, we randomly experiment with 1000 test instances and compare the speedup between Dijkstra's algorithm and our A^* search. For these over-estimate heuristic functions, we also list the number of different paths which are different from those of Dijkstra's, see table 4.1. According to our experiment, the fare and transfer heuristic functions which apply preprocessing technique of kd-tree do not outperform Dijkstra's (the speedup of most test instances are less than 1); the time heuristic function has a better performance. For details please refer to figure 4.10, figure 4.11 and figure 4.12. In our opinion, we suggest A^* search if we require the optimal solution and the major concerned is time. For these over-estimate fare functions, Fare_Manhattan_MRT (figure 4.15) and Fare_Bus (figure 4.13) obtaining different paths are both less than 10 percent. For the performance issue, function Fare_Bus does not outperform Dijkstra's but function Fare_Manhattan_MRT does. If we use Fare_MRT function (figure 4.14), it has better performance but might have more different paths (about 20 percent). According to our observation, most cases cost more 15 dollars than that of Dijkstra's algorithm. We suggest that if the price is acceptable, we can apply the heuristic functions Fare_MRT or Fare_Manhattan_MRT. For these over-estimate transfer functions, Transfer_Bus (figure 4.16) and Transfer_MRT (figure 4.17) both outperform Dijkstra's algorithm and Transfer_Bus has a better performance. Consider the path issue, these paths calculated by function Transfer_MRT almost matches that of Dijkstra's algorithm. We suggest A^* search with Transfer_MRT if user prefers optimal solution or use

Transfer_MRT function if the routing time is a major concerned.

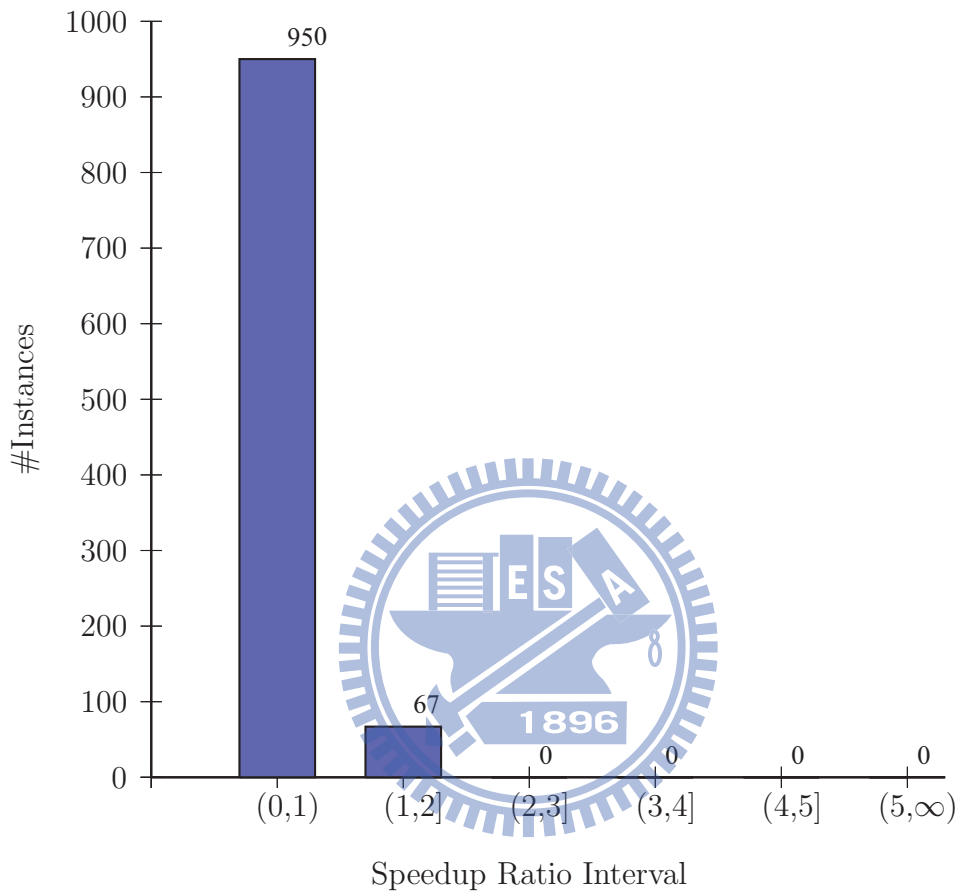


Figure 4.10: The Speedup between Dijkstra's Algorithm and A* search with heuristic function Fare by applying preprocessing technique of kd-tree

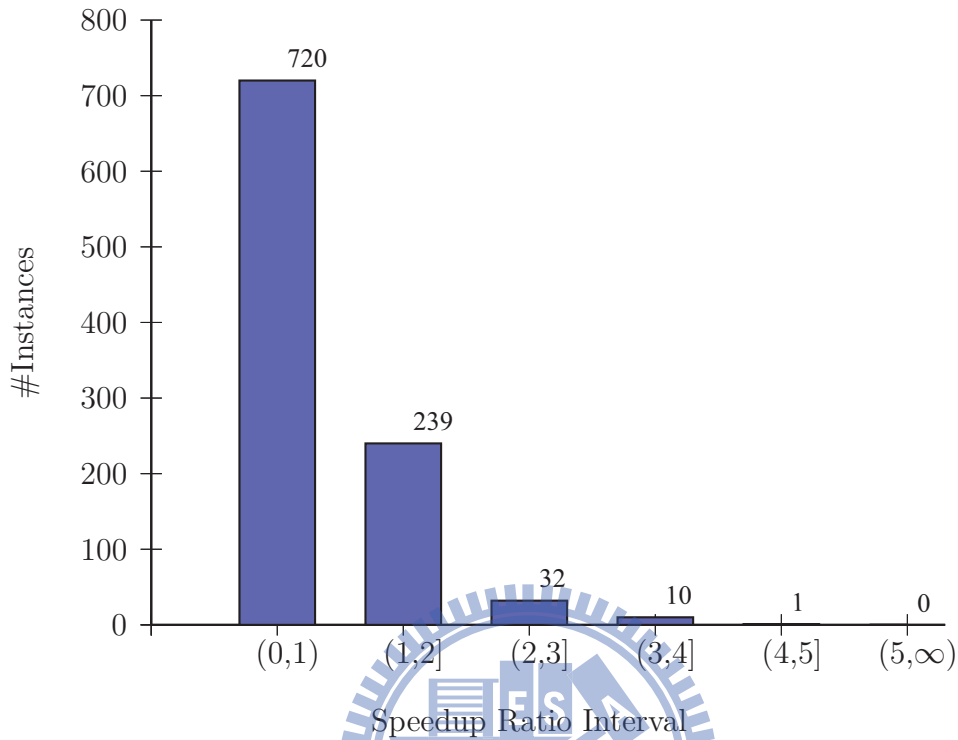


Figure 4.11: The Speedup between Dijkstra's Algorithm and A* search with heuristic function Transfer by applying preprocessing technique of kd-tree

Heuristic Function	Description	#DifferentCases per 1000 cases	Error Percentage
Fare_MRT	$\min(\Delta x, \Delta y)$ Distance and MRT Fare	214	21%
Fare_Bus	$\min(\Delta x, \Delta y)$ Distance and Bus Fare	83	8%
Fare_Manhattan_MRT	Manhattan Distance and MRT Fare	94	9%
Transfer_MRT	$\min(\Delta x, \Delta y)$ Distance and MRT Speed	9	1%
Transfer_Bus	$\min(\Delta x, \Delta y)$ Distance and Bus Speed	93	9%

Table 4.1: A table for heuristic functions whose #paths are different from those of Dijkstra's. We test 1000 cases for each heuristic function.

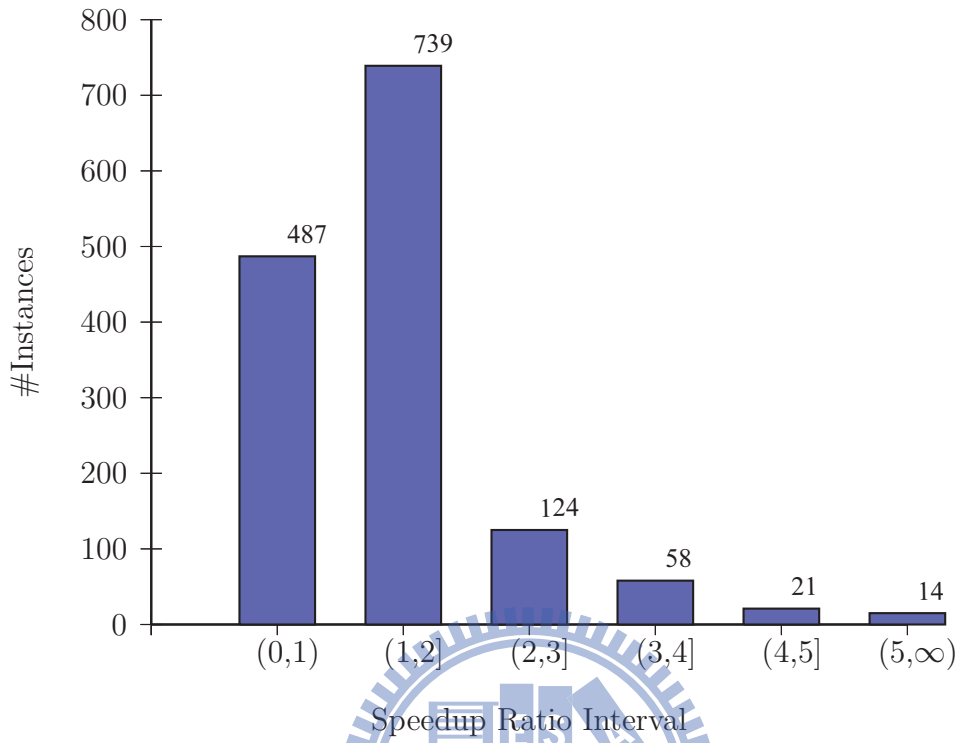


Figure 4.12: The Speedup between Dijkstra's Algorithm and A* search with heuristic function Time

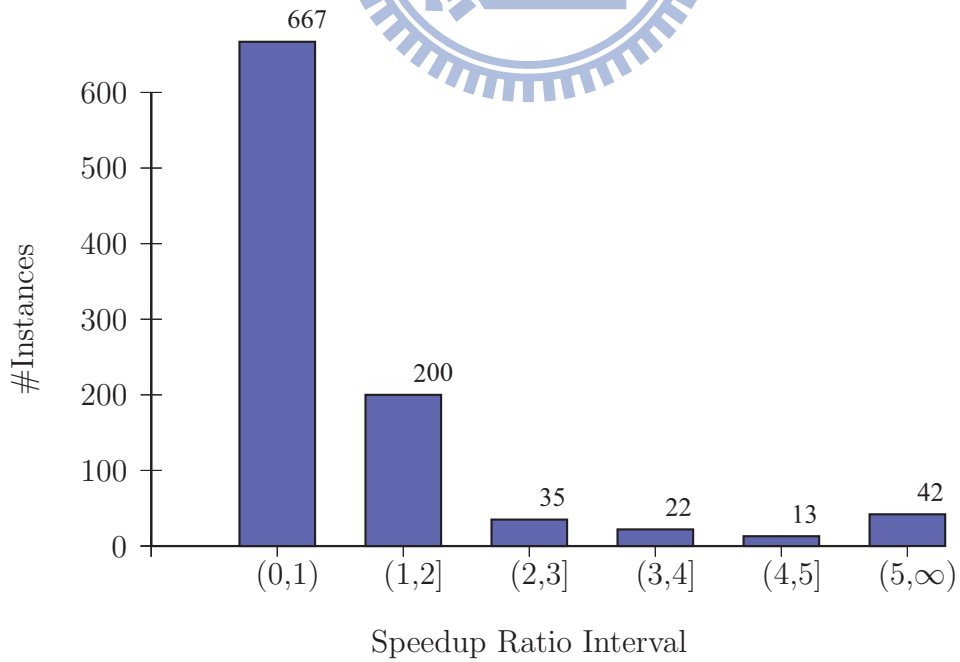


Figure 4.13: Comparison between Dijkstra's algorithm and heuristic function Fare_Bus

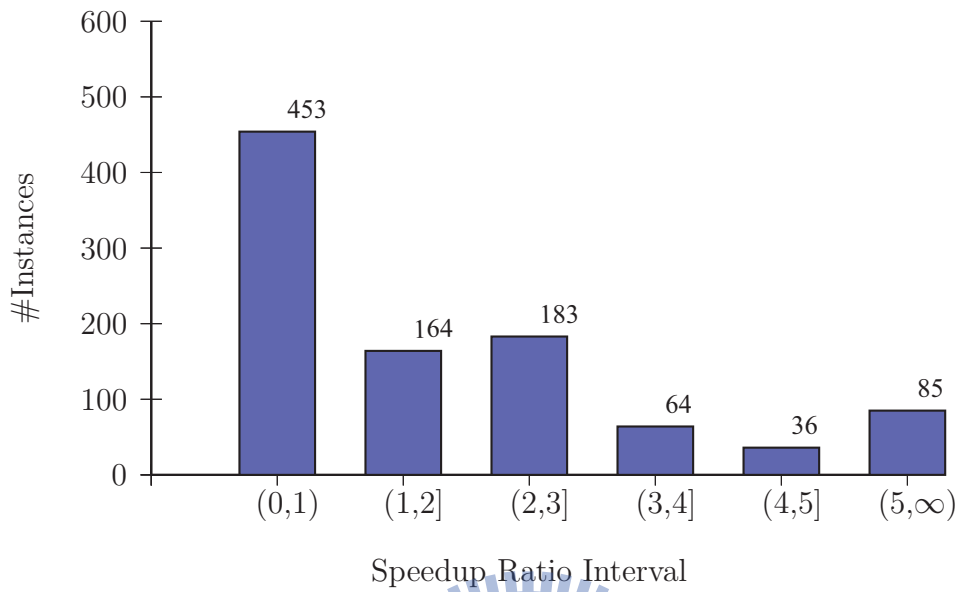


Figure 4.14: Comparison between Dijkstra's algorithm and heuristic function Fare_MRT

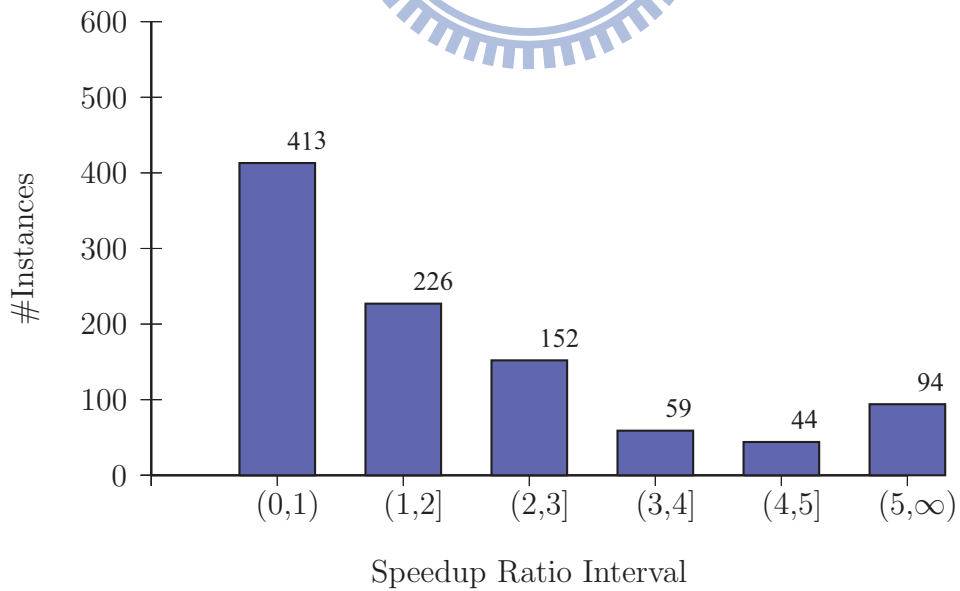


Figure 4.15: Comparison between Dijkstra's algorithm and heuristic function Fare_Manhattan_MRT

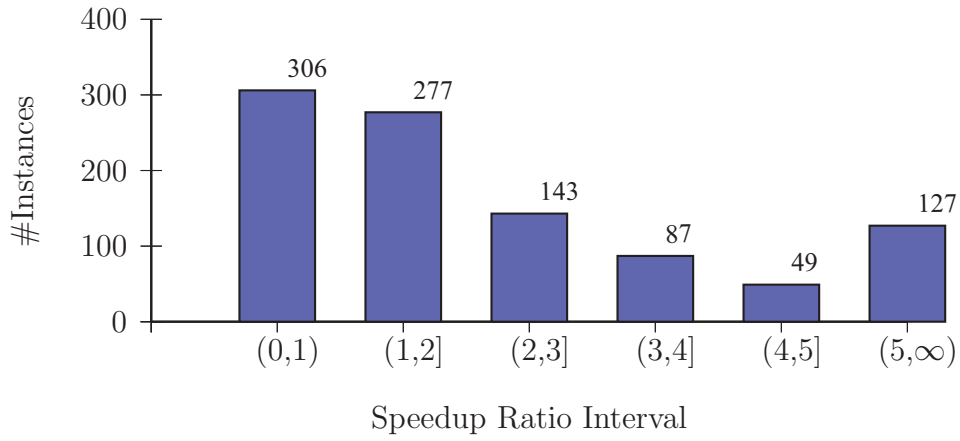


Figure 4.16: Comparison between Dijkstra's algorithm and heuristic function Transfer_Bus

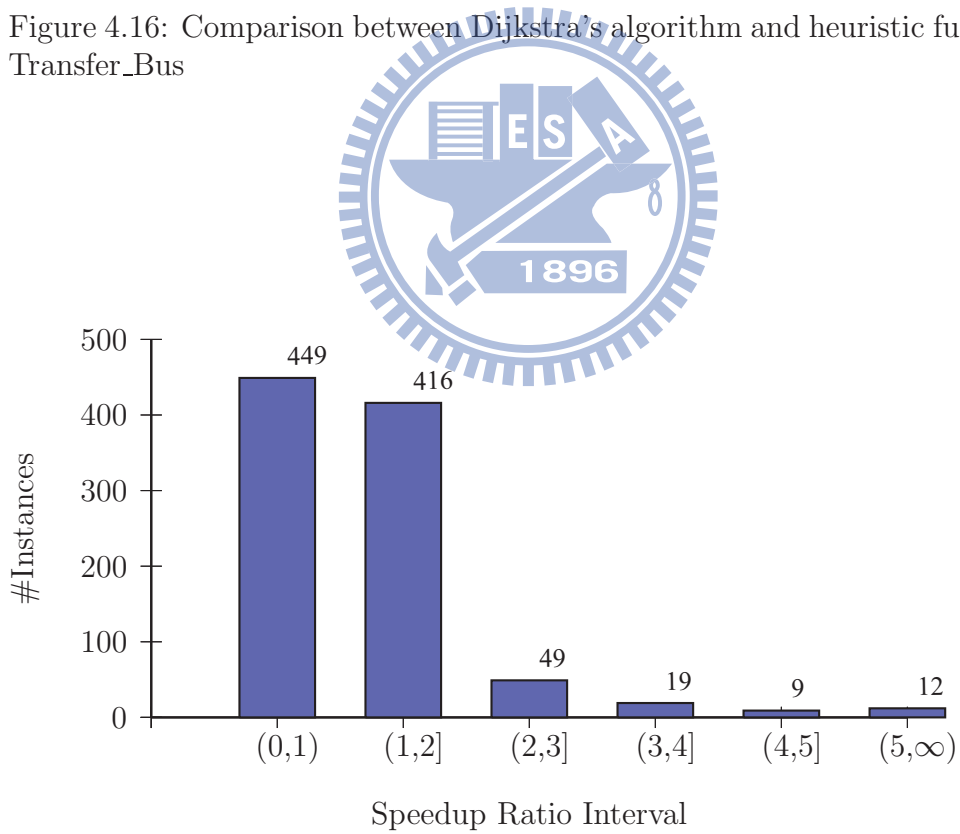


Figure 4.17: Comparison between Dijkstra's algorithm and heuristic function Transfer_MRT

Chapter 5

Conclusion

We summarize the work in this thesis. In real road network which is a sparse graphs model, we study and present a data structure “lazy heap”. According to our experiments, lazy heap can speedup the computing time about 50% over standard binary heap. In the other work, we study the public transportation system of Taipei City, and propose a route method with various metric methods. There are several unsolved problems:

- There is incomplete information in our data. Because of this, our route path may not match the real world scenario. How to get real world transportation data is a problem.
- Speedup the computing time: the bottleneck of system performance is path finding algorithm. Our idea is try to use the preprocessing technique to reduce the search time.
- Porting to small device: although our system works efficiently on the server, several critical issues are under consideration on small devices, for example, data representation or no floating point computing. How to find an efficient solution for small devices is still undergoing.

Bibliography

- [1] 9th DIMACS Implementation Challenge: Shortest Paths, <http://www.dis.uniroma1.it/~challenge9/>.
- [2] LEDA Algorithmic Solution, <http://www.algorithmic-solutions.com/>.
- [3] Kd-tree, <http://en.wikipedia.org/wiki/Kd-tree>.
- [4] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. Computational Geometry, Second Revised Edition. Springer-Verlag 2000. Section 5.3: Range Trees, pp.105-110.
- [5] D. K. Blandford, G. E. Blelloch and I. A. Kash, An Experimental Analysis of a Compact Graph Representation, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 49-61, 2004
- [6] A. V. Goldberg and R. F. Werneck, Computing Point-to-Point Shortest Paths from External Memory, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 26-40, 2005
- [7] P. Sanders, D. Schultes and C. Vetter, Mobile Route Planning, *Proceedings of the 16th annual European symposium on Algorithms*, 732-743, 2008.
- [8] L. Arge, M. A. Bender, E. D. Demaine, Holland-Minkley Bryan and I. J. Munro, Cache-oblivious priority queue and graph algorithm applications, *Proceedings of the thirty-fourth annual ACM symposium on Theory of computing*, 268-276, 2002.
- [9] P. Sanders, Fast priority queues for cached memory, *J. Exp. Algorithmics vol. 5*, 2000.

- [10] U. Meyer and V. Osipov, Design and Implementation of a Practical I/O-efficient Shortest Paths Algorithm, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 85-96, 2009.
- [11] D. Ajwani, U. Meyer and V. Osipov, Improved External Memory BFS Implementation, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 2007.
- [12] D. Wagner, T. Willhalm and C. Zaroliagis, Geometric containers for efficient shortest-path computation, *J. Exp. Algorithmics vol. 10*, 1-30, 2005.
- [13] R. H. Möhring, H. Schilling, B. Schütz, D. Wagner and T. Willhalm, Partitioning graphs to speedup Dijkstra's algorithm, *J. Exp. Algorithmics vol. 11*, 2006.
- [14] R. Bauer and D. Delling, SHARC: Fast and Robust Unidirectional Routing, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 13-26, 2008.
- [15] R. Geisberger, P. Sanders, D. Schultes and D. Delling, Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks, *6th Workshop on Experimental Algorithms*, 319-333, 2008.
- [16] A. V. Goldberg, H. Kaplan, and R. F. Werneck, Reach for A*: Efficient Point-to-Point Shortest Path Algorithms, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 129-143, 2006.
- [17] H. Bast, S. Funke, D. Matijevic, P. Sanders and D. Schultes, In Transit to Constant Time Shortest-Path Queries in Road Networks, *Proceedings of the Workshop on Algorithm Engineering and Experiments*, 2007.
- [18] P. Sanders and D. Schultes, Engineering highway hierarchies. *In Proceedings of the 14th Conference on Annual European Symposium vol 14*, 2006.
- [19] P. E. Hart, N. J. Nilsson, B. Rapheal, A formal basis for the heuristic determination of minimum cost paths, *IEEE transactions on Systems Science and Cybernetics 4*, 100-107, 1968.

- [20] E. W. Dijkstra, A note on two problems in connexion with graphs, *Numerische Mathematik 1*, 269-271, 1959.
- [21] T. Cormen, C. Leiserson, R. Rivest. Introduction to Algorithms, The MIT Press, 1990.
- [22] C. Dennis, Bidirectional Heuristic Search Again, *J. ACM vol 30*, 22-32, 1983.
- [23] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM vol 34*, 596-615, 1987.
- [24] E. Pyrga, F. Schulz, D. Wanger and C. Zaroliagis, Efficient models for timetable information in public transportation systems, *J. Exp. Algorithmics vol 12*, 1-39, 2008.
- [25] J. Tan and H. W. Leong, Least-cost path in public transportation systems with fare rebates that are path- and time-dependent, *The 7th International IEEE Conference on Intelligent Transportation Systems*, 1000-1005, 2004.
- [26] K. G. Zografos and K. N. Androutsopoulos, Algorithms for Itinerary Planning in Multimodal Transportation Networks, *IEEE Transactions on Intelligent Transportation Systems vol. 9*, 2008.
- [27] M. Thorup, On RAM Priority Queues, *SIAM J. Comput. vol 30*, 2000.
- [28] M. Thrup, Integer priority queues with decrease key in constant time and the single source shortest paths problem, *Journal of Computer and System Sciences vol. 69*, 2004.
- [29] C. Yu, A Design Platform of Personal Navigation System, *Master Thesis, College of Computer Science, National Chiao Tung University*, 2009.

Appendix A

Some Implementation Issues for PTS

We describe the data structure used for public transportation navigation system. We allocate arrays *stop*, *transport* and *neighbor*, to store the data. The information of nodes and edges are stored on array *transport*. Array *transport* is sorted by its corresponding stop id. In order to conveniently use array *neighbor* to do relaxation starting from a specific stop, we sort the array *neighbor* by (stop id, the distance to this stop) in lexicographical order. Given an index i , $stop[i]$ indicates a stop information. For index j , $stop[i].nodeStartIndex \leq j < stop[i].nodeStartIndex + stop[i].nodeCount$, $transport[j]$ indicates a transport path of stop i . Similarly, for index k , $stop[i].nbrStartIndex \leq k < stop[i].nbrStartIndex + stop[i].nbrCount$, $neighbor[k]$ indicates a neighbor stop of stop i . Function *findNeighbors* helps find the neighbor stops of current position. This is used to search start neighbor set and target neighbor set when initially user queries the route path. Function *pathFind* is used to find the transfer information; the returned value indicates whether there is any route path. Function *updateCost* calculates the cost from the current node to next node. The detail information please refer to figure A.1.

```

typedef struct TPos{
    int x;
    int y;
}TPos;

typedef struct TStop{
    TPos pos; // Position: longitude and latitude
    unsigned int transStartIndex; // the start index of corresponding transport paths
    unsigned int transCount; // the number of transport paths
    unsigned int nbrStartIndex; // its neighbors start index of neighbor list
    unsigned int nbrCount; // the number of neighbors
    char name[128]; // stop name
    unsigned char getOnOff; // only get on:0, only get off:1, both:2
} TStop;

typedef struct TStopNbr{
    unsigned int stop; // stop id
    unsigned short distance; // the distance
} TStopNbr;

typedef struct TTransport {
    unsigned int line; // pass line
    unsigned int stop; // The corresponding stop id
    unsigned int next; // its next node id
    unsigned short distance; // the distance from current node to its next node
    unsigned char time; // the travel time from current node to its next node
} TTransport;

typedef struct TValue{
    unsigned int backID; // the parent node
    unsigned int line; // by taking line to this node
    unsigned int cost; // the cost value
} TValue;

vector< unsigned int > findNeighbors(TPos* pos);
void updateCost(TValue val, unsigned int cur, unsigned int next);
BOOL8 pathFind(TPos* from, TPos* to);

```

Figure A.1: The data structures in C++ for PTS