Shortest Path Queries in Very Large Spatial Databases

by

Ning Zhang

A thesis

presented to the University of Waterloo

in fulfilment of the

thesis requirement for the degree of

Master of Mathematics

in

Computer Science

Waterloo, Ontario, Canada, 2001

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Finding the shortest paths in a graph has been studied for a long time, and there are many main memory based algorithms dealing with this problem. Among these, Dijkstra's shortest path algorithm is one of the most commonly used efficient algorithms to the non-negative graphs. Even more efficient algorithms have been developed recently for graphs with particular properties such as the weights of edges fall into a range of integer. All of the mentioned algorithms require the graph totally reside in the main memory. Howevery, for very large graphs, such as the digital maps managed by Geographic Information Systems (GIS), the requirement cannot be satisfied in most cases, so the algorithms mentioned above are not appropriate. My objective in this thesis is to design and evaluate the performance of external memory (disk-based) shortest path algorithms and data structures to solve the shortest path problem in very large digital maps. In particular the following questions are studied: What have other researchers done on the shortest path queries in very large digital maps? What could be improved on the previous works? How efficient are our new shortest paths algorithms on the digital maps, and what factors affect the efficiency? What can be done based on the algorithm?

In this thesis, we give a disk-based Dijkstra's-like algorithm to answer shortest path queries based on pre-processing information. Experiments based on our Java implementation are given to show what factors affect the running time of our algorithms.

Acknowledgements

I would like to give great thanks to Professor Edward Chan for his supervision of my thesis, and his time and energy which has been put into the project which lead directly to this thesis.

I would also like to gratefully acknowledge my appreciation for the financial support from Bell Canada and the Department of Computer Science at the University of Waterloo, which made this thesis possible at all.

My special appreciation to Professor Tamer Ozsu and Professor Timothy Chan for taking their precious time reviewing this thesis as readers, and to Sarah Vernon and Kong Ching Ma for proofreading this thesis.

Lastly, I would like to thank my friends in the University of Waterloo who helped me academically and spiritually. Most of all, I would like to thank my parents and my sister who consistently gave me confidence and support.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

1.1 The problems of route optimization on Spatial Database

In Geographical Information Systems (GIS), shortest path queries are one of the most useful and most frequently asked questions. In combinatorics, the shortest path problems on general graphs have already been well studied. For example, Dijkstra's algorithm is widely used and actually very fast when using heap data structures for priority queues [2]. Even faster algorithms are developed for graphs that have special constraints on their edge weights. For example, Cherkassky, Goldberg, and Radzik developed algorithms based on multi-level buckets [3]. The constraint of the algorithm is that the weights of the edges must be integers. With this algorithm, the time spent on searching is 1/2 to 1/3 that of Dijkstra's algorithm. However, one assumption of all of the above algorithms is that the graph can be

stored in main memory. If the digital map is too large, the algorithms cannot handle it.

Recently, several algorithms have been proposed to address this particular problem. The basic ideas are to use the divide-and-conquer method to divide the large maps into small ones, then deal with the small chunks systematically, and at last combine the solutions together. Some papers deal with the partitioning algorithms and the optimality of the solution ([1], [2]).

Some try to balance between the I/O operations and computation time [12]. In this thesis, a set of new algorithms is provided on graph partitioning and graph pruning. The materialization method proposed is also different from previous ones.

1.2 Terminology

Before proceeding to descriptions of the algorithms and the design of the system, let us examine the definitions of the frequently used terms. Terms that are not defined here are the common graph theory terms (such as vertex, edge, and path), which can be found in [3, 7].

Definition 1. (Graph)

The 3-tuple $G = (V, E, W)$ is defined to be a graph, where $V = v_i | i \in [0, n-1]$ is the set of vertices with size of n. $E = \{e_{ij} | e_{ij} =< v_i, v_j >, v_i, v_j \in V\}$ is the set of edges. Each edge is determined by a "from" vertex v_i and a "to" vertex v_j, denoted simply as e_{ij}. $W = \{w : E \rightarrow \Re^{\geq 0} | w$ is an one-to-one function from the set of edges to non-negative real numbers$\}$.

The definition of graph is actually a simple graph. The multiple edges in a graph are not considered in that we only care about the shortest paths in a graph, a multiple graph can always be simplified by removing multiple edges whose weights are not the minimum. For example, Figure 1.1 shows a typical undirected graph.

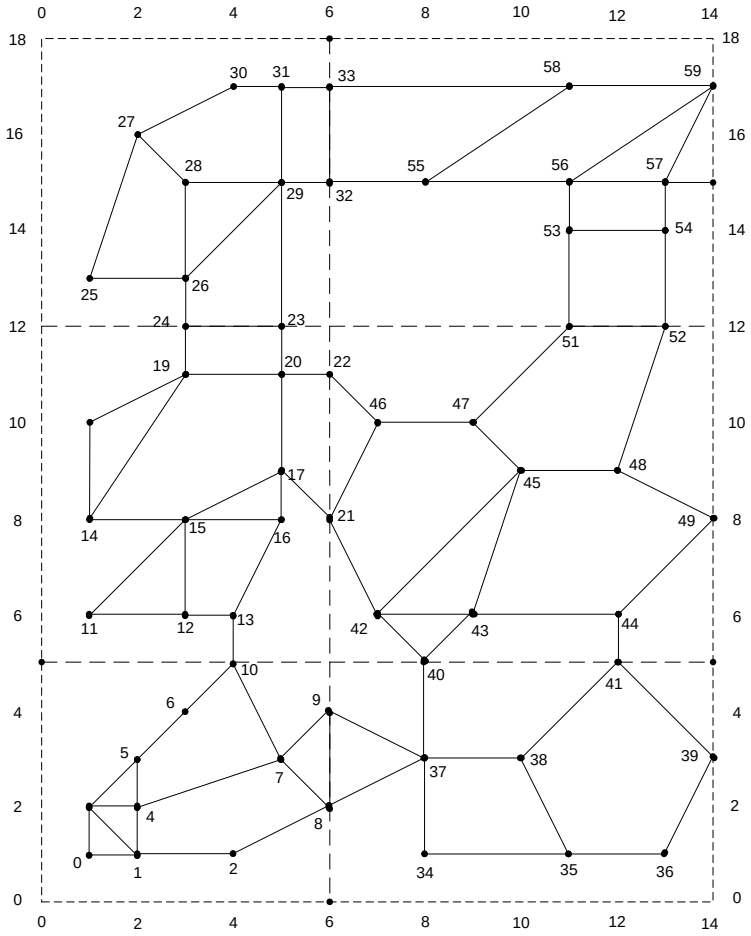

Figure 1.1: A Sample Undirected Graph

In this example, the set of vertices are labeled from 0 to 59. The edges can

be identified by the two end vertices, for example edge $< 14, 18 >$ is the edge connecting vertices 14 and 18. The weight of an edge is the Euclidean distance from one end vertex to the other end vertex. In the above example, the weight of edge $< 14, 18 >$ is 2.

By the definition of graph, there are no constraints applied on the data structure when implementing it. For convenience, the graph data structure in memory and the graph database in secondary storage are distinguished. The former is still called graph, but the latter is called "digital map". In some sense, you can think of digital map as a graph stored on secondary storage.

Definition 2. (Digital Map)

A digital map $D = (V, E, W)$ is defined to be a persistent graph on the secondary storage, where the V, E, and W are the same as defined in definition 1.

Since digital map and graph are referring to the same concept in theory, the only difference lies in the manner of implementation. When we describe the graph algorithms and other theoretical descriptions not related to the implementation, the term "graph" is used for simplicity.

Definition 3. (Sub-graph)

A sub-graph $S = (V_s, E_s, W_s)$ of graph $G = (V, E, W)$ has the following properties: $V_s \subseteq V$, and $exists$ three one-to-one functions $f_v : V_s \to V$, $f_e : E_s \to E$, $f_w : W_s \to W$ such that $\forall e_{ij} \in E_s, f_e(e_{ij}) = (f_v(v_i), f_v(v_j)), f_w(w_s(e_{ij})) = w(f_e(e_{ij}))$.

By the definition of sub-graph, the vertices in the sub-graph are a subset of the vertices in the original graph. There is an edge connecting two vertices in the sub-graph only if the two corresponding vertices in the original graph are adjacent. The edge weights in the sub-graph are the same as those of corresponding edges in the original graph. For example, a sub-graph of the graph in Figure 1.1 can be constructed. The vertices of the sub-graph are $0, 1, 3, 4, 5$, the edges of the sub-graph are $< 0, 1 >$, $< 1, 4 >$, $< 0, 3 >$, and $< 3, 5 >$. The weights of the edges are the same as those in the original graph. Note that although $< 1, 3 >$ and $< 3, 4 >$ are also edges in the original graph connecting the vertices in the sub-graph, it is not necessary to include them in the sub-graph, which is different from the definition of a fragment.

Definition 4. (Fragment)

Fragment $F = (V_f, E_f, W_f)$ is a connected sub-graph of $G = (V, E, W)$, where $V_f \subseteq V$, and $\forall e_{ij} \in E_f \Rightarrow f_e(e_{ij}) \in E$, and $\forall e_{ij} \in E \wedge f_v^{-1}(v_i), f_v^{-1}(v_j) \in V_f \Rightarrow f_e^{-1}(e_{ij}) \in E_f$. The weight of the edge in the fragment is the weight of the corresponding edge in the original graph. That is, $\forall e_{ij} \in E_f, w_f(e_{ij}) = w(f_v(v_i), f_v(v_j))$.

A fragment is a special kind of sub-graph with the following properties:

1. A fragment is a connected component. For undirected graphs, it is a complete graph, i.e. every pair of vertices has a path connecting them.

2. There exists an edge connecting two vertices in a fragment if, and only if, the

two corresponding vertices in the original graph are adjacent.

For example, in Figure 1.1, the original graph can be divided into six fragments indicated by the dashed lines. Each fragment is a sub-graph of the original graph and when the six fragments are merged together, we get the original graph. Note that the vertices 23 and 24 are shared by two fragments (which are called boundary vertices as defined later), so the edge $(23, 24)$ is also shared by the two fragments by the property 2. Later it will be seen that the edges connecting boundary vertices of the same fragments need not satisfy the property 2 in order to get the correct shortest path in our algorithm. That is the fragments satisfy the relaxed property:

2′ If both vertices u and v are connected in the original graph and are contained in k fragments. The edge (u, v) can be in any one or many of these k fragments.

If a fragment satisfies property 1 and 2′ , we call it "rimless fragment". Since rimless fragment is the same as fragment in terms of shortest paths, sometimes they are not distinguished.

Definition 5. (Partition)

A partition of a graph $G(V, E, W)$ is a set of fragments $\{F_i = (V_i, E_i, W_i) | i \in [0, n-1], \bigcup V_i = V\}$.

By definition of fragment, edges are "copied" from the original graph to the fragment if both end vertices are in the fragment. Therefore, for a partition of a

graph, we can also get the conclusion that $\bigcup E_i = E$, $\bigcup W_i = W$. This is also true for rimless fragments. In the example of Figure 1.1, the partition is the six fragments.

Definition 6. (Interior Vertex, Boundary Vertex)

Vertices in a fragment $F = (V_f, E_f, W_f)$ of graph $G = (V, E, W)$ can be divided into two sets: V_i and V_b, where $V_f = V_i \bigcup V_b$. A vertex in fragment $v_i \in V_b \Leftrightarrow \exists$ an adjacent vertex u of $f_v(v_i) \in V$ such that there does not exist a vertex v_j in V_b such that $f_v(v_j) = u$. That is, every boundary vertex connects to at least two fragments of its partition. Vertices in V_b are called boundary vertices. Any other vertices in V_i are called interior vertices.

Intuitively, boundary vertices are vertices that appear in more than one fragment, and interior vertices are vertices appear in only one fragment. Based on the definition, we can get the following properties of boundary vertex and interior vertex:

1. $\forall v_i \in V_i, degree(v_i) = degree(f_v(v_i))$.
2. $\forall v_j \in V_b, degree(v_j) < degree(f_v(v_j))$.

The two properties of boundary vertices and interior vertices can be obtained easily from the definition of fragment. The first property implies that an interior vertex is only adjacent to the interior vertices of its own fragment or the boundary vertices of its own fragment of its adjacent fragments. That is, there are no edges connecting an interior vertex in one fragment to an interior vertex in another

fragment. It follows that a path connecting an interior vertex with a vertex in another fragment must pass through one or more boundary vertices. Note that, by definition, the set of interior vertices could be empty, but in practice, in order to get better performance, the set of interior vertices is always non-empty. Otherwise, we cannot localize the program by partitioning the graph into fragments.

Also taking Figure 1.1 as an example, we can get the original graph when we merge the six fragments. For the fragment at the lower left corner, interior vertices are 0, 1, 2, 3, 4, 5, 6, and 7. The boundary vertices are 8, 9 and 10. All edges in the original graph are "copied" to the fragment, and they are the only edges present in the fragment. The edge weights remain the same with the original graph. If there are edges connecting two boundary vertices in the original graph, the edges also appear in those fragments, such as the edge ¡8,9¿ appears in both the lower left and lower right fragments.

Definition 7. (Boundary Set)

A boundary set is the set of all boundary vertices shared by two or more fragments. A boundary set can be denoted by $BS[f_i, f_j, \ldots, f_k]$, where $f_i, f_j, \ldots, f_k$ are the fragments that share the boundary vertices in the boundary set. $f_i, f_j, \ldots, f_k$ are sorted in ascending order so that the sequence of $f_i, f_j, \ldots, f_k$ can uniquely determine the boundary set. The sequence is also called the ID of the boundary set.

The idea behind the concept of boundary set is that the boundary vertices in

a boundary set are shared by the same set of fragments, so a boundary set can be contracted into one super vertex when one is interested only in the connectivity of boundary sets. Therefore, in terms of fragment connectivity, boundary set acts as an equivalence set. This nice property is used in constructing sketch graphs.

Each fragment may have zero or more boundary sets. A boundary vertex can be in two different boundary sets. If this is the case, the boundary vertex must be shared by more than two fragments. For example, in Figure 1.1, we have six fragments. We can name them by integers, say 0, 1, 2, 3, 4, and 5 from left to right, bottom to up. Therefore, the lower left is fragment 0, the lower right is fragment 1, the upper right is fragment 5, and so on. Seven boundary sets can be found in this partition, namely $BS[0,1]$, $BS[0,2]$,$BS[1,3]$, $BS[2,3]$, $BS[2,4]$, $BS[3,5]$, and $BS[4,5]$, where $BS[0,1] = \{8,9\}$, $BS[0,2] = \{10\}$, $BS[1,3] = \{40,41\}$, $BS[2,3] = \{21,22\}$, $BS[2,4] = \{23,24\}$, $BS[3,5] = \{51,52\}$, and $BS[4,5] = \{32,33\}$.

Note that a boundary vertex could be in multiple fragments. If boundary vertex is allowed to appear in different boundary sets, the boundary sets can be restricted to be boundary sets between two fragments. For example, if a boundary vertex is in fragment 1, 2 and 3, it should be in the boundary sets $[1,2]$, $[2,3]$ and $[1,3]$. (Actually, the result of my thesis is that all the boundary sets are boundary sets between two fragments. The reasons to do this are 1) there are few boundary vertices in more than two boundary sets. 2) finding boundary vertices between two fragments is much easier than doing so among three or more fragments.)

Definition 8. (Super Graph)

A super graph $S = (V_s, E_s, W_s)$ of a graph partition $F_1, F_2, \ldots F_n$ has the following properties: $V_s = \{v_b | v_b$ is boundary vertex in $F_i, i \in [1,n]\}$, $E_s = \{(v_i, v_j) | \exists F_k, v_i, v_j \in V_k\}$ $W_s = \{w_s(e_{ij}) | w_s(e_{ij}) = \min(\{SD_k(e_{ij}) | k \in [1,n]\})\}$ where SD_k is the shortest distance function from v_i to v_j in fragment F_k, min is the minimum function, if v_i and v_j are not connected in F_k, $SD_k(e_{ij}) = \infty$.

The super graph of a partition can be thought of as a graph consisting of one complete sub-graph for each fragment. The vertices of the super graph are the boundary vertices in the fragment. The edge weights of the sub-graph are the minimum of shortest distances in the all sub-graphs containing the two end vertices, or infinity if no paths connect them. An example of the super graph of Figure 1.1 is illustrated in Figure 1.2.

In this example, only boundary vertices are included in the super graph, and for each pair of boundary vertices in the same fragment, there is an edge connecting them. The weights of the edges are the shortest distance inside the fragment from one boundary vertex to the other boundary vertex.

Definition 9. (α-value, β-value)

The α-value of a set of vertices S to a set of vertices D in graph G is the minimum value of the shortest distances from any vertex $v \in S$ to any vertex $u \in D$. It can be written as $\alpha(S, D) = \min(\{SD(v, u) | v \in S, u \in D\})$. Similarly, the β-value of a set of vertices S to D can be written as $\beta(S, D) = \max(\{SD(v, u) | v \in S, u \in D\})$.

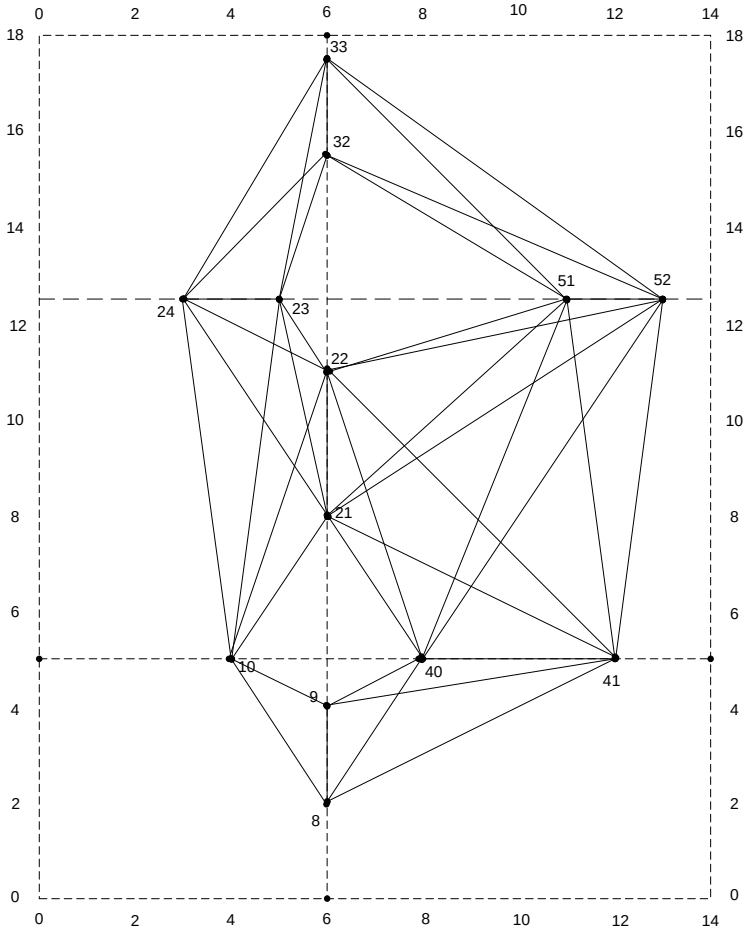

Figure 1.2: Super Graph

Definition 10. (Sketch Graph, α-graph, β-graph)

A sketch graph $S = (V_s, E_s, W_s)$ of a graph partition $\{F_1, F_2, \ldots, F_n\}$ has the following properties: $V_s = \{v_s | v_s$ corresponds to some boundary set in $F_i\}$, that is $\exists$ a bijection f, where BS_i is the set of boundary sets in the i^{th} fragment F_i. $E_s = \{(v_i, v_j) | \exists F_k, f(v_i) \subseteq V_k\}$, where f is the bijection defined in V_s $W_s = \{w_s : E_s \rightarrow (\Re^{\geq 0}, \Re^{\geq 0})\}$, where w_s is an one-to-one function from the set

of edges to a set of 2-tuple (α, β), where α and β are the α-value and β-value for the two corresponding boundary sets in the super graph respectively. α-graph is a sketch graph, but the weights of edges are the α-value of the two boundary sets in super graph, instead of the 2-tuple (α, β). Similarly, β-graph is a sketch graph with the β-values as edge weights. The shortest distance from s to d in the α-graph and β-value are denoted as $SD_\alpha(s, d)$ and $SD_\beta(s, d)$ respectively.

The sketch graph carries a high level outline of the partition, describing the connectivity of boundary sets and what might be the possible shortest distance from one boundary set to another. This information can be used to prune a super graph to get a super graph. The sketch graph of example of Figure 1.1 is shown in Figure 1.3.

In this sketch graph, there are only seven vertices corresponding to seven boundary sets in the partition. Each pair of boundary sets in the same fragment has an edge connecting them. The α-value and β-value are also labeled on the sketch graph.

Definition 11. (Hierarchical Graph)

A hierarchical graph of a graph G is defined by $H = \{P_0, P_1, \ldots, P_n\}$, where P_0 is the partition of the ground-level graph, P_1 is the partition for the super graph based on P_0, P_i is the partition for the super graph based on P_{i-1}, and so on.

In the hierarchical graph decomposition principle, the ground level graph is partitioned into small fragments first. Then the super graph is built on top of the

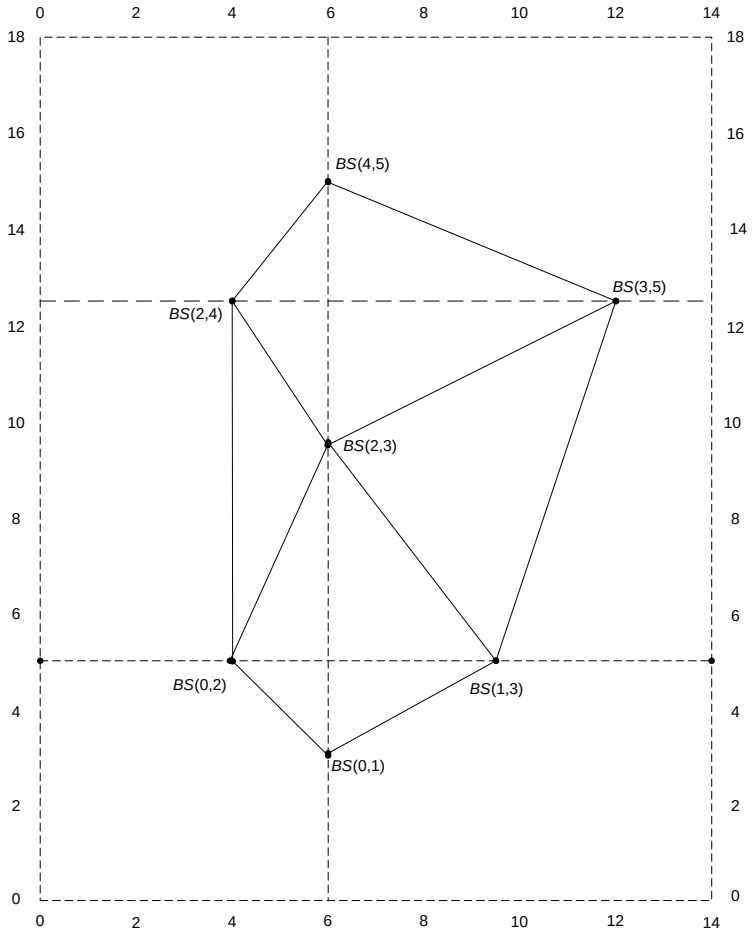

Figure 1.3: Sketch Graph

partition. The super graph is further partitioned to generate even higher level super graphs until the top-level super graph is small enough, i.e. suitable for conquering. There are problems associated with this approach, which will be seen when the SPC partitioning algorithm and HEPV approaches are analyzed.

Chapter 2

Survey on Previous Works

2.1 Fundamental Works

In this section, the previous fundamental works in the shortest path algorithms are introduced. Four typical algorithms or approaches are selected:

- Dijkstra's shortest path algorithm.
- $A*$ search heuristics.
- Lipton and Tarjan's planar graph separator algorithm.
- Frederickson's all-pair shortest path algorithm on planar graphs based on hammock decomposition.

These algorithms are of particular interest because they have solved basic problems (small sized graphs) very well and gave inspiration for the new algorithms on

extended problems. They still have limitations so we should not rely on them totally. For example, both Lipton and Tarjan's planar graph separator algorithm and Frederickson's hammock decomposition take planar graphs as partitioning input. However, sometimes a partitioning algorithm is needed for very large non-planar graphs, so a new graph partitioning algorithm should be developed, although it may not be as optimal as the two algorithms mentioned above. The three fundamental algorithms are introduced first. Then in Section 2.1.2, solutions for shortest paths on very large graphs are discussed by means of divide-and-conquer approach.

2.1.1 Shortest Path Algorithm

The shortest path problem has been studied for a long time and the most frequently used a general-purpose algorithm for non-negative graphs is the Dijkstra's shortest path algorithm [1]. Almost all of the algorithms we will discuss later, which are trying to tackle the shortest path problems on special property graphs, are based on the Dijkstra's algorithm, here is a quick look at it first. Dijkstra's algorithm is a best-first search greedy algorithm. It performs a search of the graph from the source node s in iterations. Vertices in the graph can be divided into two classes:

- The shortest distance from s is already known. These vertices are called close vertices.
- The shortest distance from s is not known, but they may have a distance (candidate shortest distance) associated with them. These vertices are called open vertices.

In each iteration, one open vertex with the minimum distance is selected and closed. Then shortest distances are updated for all open neighbors of the newly closed vertex. This process is called relaxation. The distances can be maintained in a priority queue, which is usually implemented by heaps. Given the $O(\log n)$ complexity of updating heaps, and $O(n)$ updates in handling a planar graph, the time complexity of constructing a shortest path tree is $O(n \log n)$ [1]. This time complexity is for a single source problem. If we want all-pairs shortest paths, Dijkstra's algorithm need to be executed n times, so the time complexity is $O(n^2 \log n)$ for planar graphs.

2.1.2 $A*$ Search Heuristics

$A*$ search, like Dijkstra's algorithm, is a best-first search technique [2], but it uses a heuristic function $h(v)$ to calculate the estimate cost from a vertex v on the path to the destination vertex d. This heuristic function, combined with another function $g(v)$ – the cost from the source vertex s to v, determines the evaluation function $f(v)$, i.e. $f(v) = f'(g(v), h(v))$. Usually f' simply performs a sum of $g(v)$ and $h(v)$, so $f(v) = g(v) + h(v)$. The value of function represents the estimated cost of the solution through vertex v. The algorithm using $A*$ search is described as follows: At the beginning, all vertices are open except the source vertex s. Starting from s, for each of the open neighbor v of s, calculate the function $g(v)$ and $h(v)$, as well as the evaluation function $f(v)$ based on $g(v)$ and $h(v)$. Then choose the vertex with the minimum value of f-value, say u, to be the next vertex on the shortest path tree and mark it as closed. Restart the same procedure as above from u until the destination vertex is closed.

Usually $g(v)$ is simply the distance from s to v, so if we do not take $h(v)$ as a parameter of evaluation function $f(v)$, $A*$ search is actually the same as Dijkstra's shortest path algorithm. It is this heuristic function $h(v)$ that let $A*$ search stand out from other algorithms. This heuristic function can be embedded in the algorithm to represent some constraints enforced to the queries. For some constraints, such as "the solution must not pass through a certain street" or "the solution must not go to eastward", it can be specified by heuristic functions easily. Some may not. For example, if the query demands that the path must go through an edge, you cannot simply label the edge cost to be a very small value or even $-\infty$, since it cannot guarantee that the search will reach one of the vertices on the edge in the first hand. There is no simple way to express such constraints. However, for the constraints that can be expressed by heuristic function, $A*$ search is a natural way. For example, take into consideration the constraint, "the solution must not pass through a certain street". Although you can also set the edge cost to be ∞ in Dijkstra's algorithm, it cannot be maintained as easily as using $A*$ search, since the edge costs must be set before the algorithm starts off for Dijkstra's algorithm, whereas the calculation of $h(v)$ is online for $A*$ search. So, in the cases when the query has constraints like "I do not want to pass streets with too many stop signs", for Dijkstra's algorithm, the number of stop signs in each of the street in the map must be counted, and an appropriate weight on the edge cost must be assigned. But for $A*$ search, only the stop signs on the streets which are incident to the vertices examined need to be counted. Therefore, other streets do not have to be calculated. This may save much time when the source and destination vertices are

very close and the map is very large. Another advantage of $A*$ search is that it does not only allow embedding of constraints on queries, but also saves time on the searching phase especially when the searching space is very large. For example, if the coordinates of the source and destination vertices are known, the Euclidean distances between any vertex to the destination vertex should be known. The $A*$ heuristics can take this distance as a guide to decide which vertex looks like the next closed vertex.

The problem of $A*$ search is that the heuristic function is not easy to generate dynamically. It is prefered that the heuristic function be generated online because in online applications such as route-planning systems, queries and constraints are given by the end users. For each set of constraints, there should be a different heuristic function with which it is associated. Determining the heuristic function dynamically based on the user's queries is not trivial. For some most frequently specified constraints, such as put some "barriers" on the map, a pre-determined heuristic function can be embedded in the algorithm.

2.1.3 Lipton and Tarjan's Planar Graph Separator Algorithm

In [34], Lipton and Tarjan gave a separator algorithm for planar graphs, which guarantees the upper bound of the ratio of boundary vertices to interior vertices. This algorithm is based on a theorem they gave in the same paper that any n-vertex planar graph can be partitioned into three sets A, B and C, such that no edges joins a vertex in A and a vertex in B, neither A nor B has more than $2n/3$ vertices, and

C contains no more than $2\sqrt{2n}$ vertices. In the shortest path problem, A and B are the two subgraphs (fragments) divided by the separator C (boundary vertices). The algorithm based on this theorem can find such sub-graphs and separator in $O(n)$ time. The concept behind the algorithm is as follows.

In order to divide vertices in the graph into two distinct partitions without vertices in one partition having edges joining vertices in the other partition, the vertices must be partitioned in a way such that the vertices in one class only connect to vertices in the same class or to a limited number of other classes. This property can be achieved by breadth-first spanning tree. The breadth-first spanning tree is obtained as follows: choose an arbitrary vertex as root and perform breadth-first search traversing the whole graph. The edges traversed the bread-first search construct a tree structure. The vertices in the tree can be partitioned into levels according to the distance from the root. The root vertex is the only vertex in level 0, the children of root is in level 1, and the children of vertices in level 1 is in level 2 and so on, until the bottom of the tree is reached. The height of the tree is called em radius of the spanning tree. For example, the graph in Figure 2.1(a) can be represented by the breadth-first spanning tree in Figure 2.1(b).

By properties of breadth-first search, it is known that all edges in the graph are only within level l or from l to $l+1$. That is there is no edge in the graph connecting a vertex in level $l-1$ and a vertex in level $l+1$, otherwise the vertex in the level $l+1$ should be in level l, rather than $l+1$. Lipton and Tarjan ([34]) then gave an important proof for the statement: there exists a separator C of size no more than $2r+1$ (where r is the radius of the BFS spanning tree) such that C

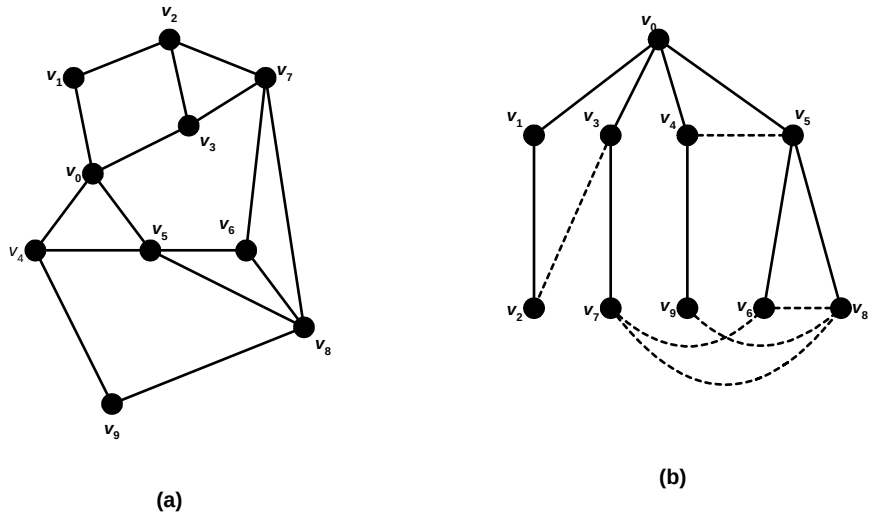

Figure 2.1: (a) An undirected graph. (b) The corresponding breadth-first spanning tree rooted at v_0. v_0 is in level 0; v_1, v_3, v_4 and v_5 are in level 1; v_2, v_7, v_9, v_6 and v_8 are in level 2. The radius of spanning tree is 2.

separates the vertices in the planar graph into two sets A and B, neither of which has cost more than 2/3. Then the separator theorem can be easily extended from this statement and other lemmas. For detailed proofs, please see [34].

This separator theorem along with the $O(n)$ algorithm thereof given by the authors guarantee the upper bound of a planar vertex separator, and getting it efficiently. However, the separator theorem is only available for planar graphs. For applications that cannot guarantee planarity, it cannot be used directly. However, it is inspiring that the underlying BFS algorithm may be the right way to separate common graphs.

2.1.4 Frederickson's Hammock Decomposition

Frederickson's hammock decomposition [26, 31, 32, 33] is not used in any of the algorithm discussed later, but due to the beautiful properties of hammocks and its close relation to those algorithms discussed, a brief introduction here can be used as a very good example for comparisons with other algorithms.

A hammock decomposition algorithm decomposes a planar graph into outerplanar subgraphs, with the properties that at most, four vertices in each hammock are boundary vertices, that are shared with the rest of the graph. The two nice properties of hammocks, outerplanarity and bounded number of boundary vertices, yield very promising results on the shortest path problem. Firstly, since the number of boundary vertices are bounded, the possible number of routes from an interior vertex in one subgraph to a vertex in another subgraph is bounded, i.e. there are only $4 * 4 = 16$ possible routes. Therefore, it is very fast to determine the shortest one in the 16 routes. Secondly, since the hammocks are outerplanar graphs, vertices can be labeled in a clockwise order around some faces which generates the hammocks. Because the neighbors have the similar route from the source vertex, vertices can be grouped into equivalence classes in terms of the shortest path. That is, given a source vertex in the outerplanar graph, and supposing its out degree is n, then the other vertices in the outerplanar graph can be partitioned into at most n equivalence classes. Vertices are in the same equivalence class if, and only if, the shortest paths from the source vertex to these vertices start off from the same out edge. For edges with no negative costs, it can be proved that each equivalence class is a set of intervals according to the numbering of the vertices around their

faces. The routing table can take advantage of this property and does not need to encode every particular vertex in the routing table; instead it only needs to encode the intervals, which will reduce the amount of storage for the routing table significantly.

The limitation of Frederickson's algorithm is that the input graph must be planar, which may not be satisfied in many applications. Even if the input graph is planar, the decomposed subgraphs may very small so the number of boundary vertices is very large. For example, if the planar graph is a grid graph, the hammock decomposition results $O(n)$ subgraphs, each of which is a grid in the original graph. This property reduces its usability in application such as road planning systems, since many road systems have grid-like feature.

2.2 Previous Disk-based Shortest Path Algorithms

Normal internal-memory algorithms can perform terribly when the problem instances get large [40], because RAM-complexity and I/O-complexity play different roles in different problem instances. Disk-based (a.k.a. external-memory) algorithms and data structures are the way around this limitation. An example of this extension can be found in a B-tree, which is an disk-based extension of the internal memory data structure k-nary tree. In the rest of this chapter, the two disk-based shortest path algorithms based on divide-and-conquer approach will be surveyed. The first is presented in [10, 11, 12, 13, 14] proposed by Jing, Huang and Rundenstene, and was later studied by Shekhar, Fetterer, Goyal, Kohli, and Coyle. They use a Spatial Partition Clustering (SPC) algorithm for partitioning the digital map

and construct routing tables for auxiliary external data structure for shortest path queries. The second algorithms is presented in [41, 42] proposed by Hutchinson, Maheshwari, and Zeh which use an external-memory extension of a planar graph separator algorithm (see [34]) to partition the digital map and external-memory rooted trees for shortest path queries. Although they use different particular algorithms for partitioning and query answering problems, they share the same design idea: divide-and-conquer.

2.2.1 General Ideas

By divide-and-conquer, the whole algorithm can be divided into two phases: pre-processing phase and query processing phase. In the pre-processing phase, the digital map is divided into sufficiently small fragments which are then stored in disk-based data structures. Different algorithms do different pre-computation and thus store different information in disk-based data structures. No matter what the information stored, the goal of materialization is the same: reduce the computation in the querying phase by reading from the disk-based data structure. Thus, how to organize the data in the disk-based data structure to access the information more efficiently is an important issue. Both algorithms organize the pre-computed information into a hierarchical tree structure so that traveling down from root to a leaf node need only small number of I/O. In the query processing, both algorithms need to read data from the tree structure and do exhaustive comparisons on candidate shortest paths. The path with the minimum value is recorded and returned as the shortest distance. The construction of the shortest path depends on the particular

algorithm. In the next two sections, the two algorithms are introduced in more detail.

2.2.2 Routing Table Method

The routing table method is proposed in [10, 12, 13] for storing the shortest path information in a routing table as those used in the computer network routers. The method first partitions the digital map into small fragments, and then boundary vertices are pushed to the second level to form a super graph. All-pair shortest paths among the boundary vertices in the same fragment are also performed in each fragment, and the corresponding edges are added to the super graph. If the super graph is still too large, it is divided further into fragments and a third level super graph is generated. This process goes on until the top-level super graph is small enough. This whole set of super graphs and the ground level graph is called a hierarchical graph. The all-pair shortest paths pre-computations at each level are stored in routing tables.

After the pre-processing phase, the system is ready to accept the shortest path queries. Given the source s and destination d, the systems should first look for the fragments containing the two vertices, say S and D respectively. S and D are either (a) in the same fragment, or (b) in different fragments. In the case (a), the shortest path could be totally in the fragment, or part of it could be in other fragments, thus the path must pass through some boundary vertices of this fragment. In the case (b), the shortest path connecting two different fragments must pass through some boundary vertices in each fragment. Therefore, a common operation of both

cases is as follows:

1. Compute the shortest paths from s to all boundary vertices in the S ($s \rightsquigarrow BV(S)$) and those from all boundary vertices in D to d ($BV(D) \rightsquigarrow d$).

2. Compute all possible combinations of $s \rightsquigarrow BV(S) \rightsquigarrow BV(D) \rightsquigarrow d$, and find the minimum one.

In the case (b), the minimum one is the final answer. However, in the case (a), a shortest path search must be done inside the fragment S and compare the distance with the minimum value found in step 2. The lessen is the final answer.

Pre-processing Phase: Spatial Partitioning Cluster (SPC)

In [10], Huang, Jing and Rundensteiner gave a heuristic algorithm on how to partition the digital map considering the following characteristics of a GIS road map:

1. GIS maps are relatively sparse, and fan out usually between 2 and 5.

2. GIS maps are strongly connected, with each node typically reachable from near-by nodes in a few hops.

3. GIS maps consist of mostly short links comparing to their map size.

The Spatial Partition Clustering (SPC) takes these characteristics into consideration, and the goal is to achieve I/O optimization in path query processing. The SPC algorithm uses a plane sweep technique based on the order of coordinates and is trying to get the fragment as square as possible. The algorithms works like this:

1. Sort all links by the x-coordinates of their origin nodes.

2. The sweeping process stops periodically to sort the links swept since the last stoppage to sort the links by y-coordinates of their origin nodes.

3. After each y-sort, the y-sorted links can be grouped into pages and written to a new linked table that is SPC clustered.

The tricky part of the algorithm is to determine when to stop sweeping and start y-sorting. Their heuristic is: the road information is stored in a link table at first. The output would be a clustered link table in which adjacent links in a square-like region are grouped into blocks. It maintains a temporary block table to store the sorted links so far, and keeps track of three parameters: dx_i indicates the difference between the minimum and maximum x-coordinate values of the original nodes in the block table, dy_i indicates the difference between the minimum and maximum y-coordinate values of the original nodes in the block table, p indicates how many pages of link table has been written to the block table. The algorithm first reads one page of x-coordinate sorted link table to the block table and increases p by 1. When there is more than one page in the block table, computer d_p and d_{p-1} , where $d_p = |(dy_p/p) - dx_p|, d_{p-1} = |(dy_{p-1}/(p-1)) - dx_{p-1}|$. If $d_p > d_{p-1}$, this is a stoppage and the links in block table should be sorted by their y-coordinates and append them to the final clustered link table. Since there are exactly p pages in the block table, the clustered link table is also extended by p pages. All of the p pages are in a certain range of x-coordinates, and the y-coordinates of the links in the first page are less or equal to the y-coordinates of those in the second page, and so on. Therefore, each page resembles a square region. The idea behind the heuristic is that when the first few pages are written to the block table, p and dx_p

are small, so $d_p = |(dy_p/p) - dx_p|$ is likely to be large. When more pages are read, p and dx_p will get larger, thus d_p should get smaller, chances of dp being greater than d_{p-1} will get larger, where the stoppage will happen.

The heuristic works fine when the flat map resembles a square or cycle, since in such situations, the dy_p does not change greatly, so the value of d_p changes as we expect. When the flat map is in some particular shapes, such as a strip whose d_x is much greater or less than d_y, the partitions based on SPC are not satisfactory.

Query Answering Phase: Exhaustive Comparative Shortest Path Algorithm

After getting the partitions of the ground level graph, we can find the boundary vertices for each fragment and compute the all-pair shortest paths among boundary vertices, then the hierarchical graph can be generated bottom-up. A super graph (non ground level graph) in the hierarchical graph consists of only the boundary vertices of the next lower level graph (super or ground level graph), and there is an edge in the super graph if the two vertices are in the same fragment. The weight of the edge is the shortest distance from one vertex to the other. When the source and destination vertices are provided, the algorithm enumerates all possible combinations of boundary vertices in the source fragment and those in the destination fragment as shown in Figure 2.2.

The shortest path from s to d must be a combination of a shortest path from s to a binary vertex u in S, the shortest path from u to a boundary vertex v in D, and shortest path from v to d, i.e., $SD(s,d) = \min\{SD(s,u) + SD(u,v) + SD(v,d)\}$ $(*)$, where u and v are any boundary vertices in the fragment S and D respectively.

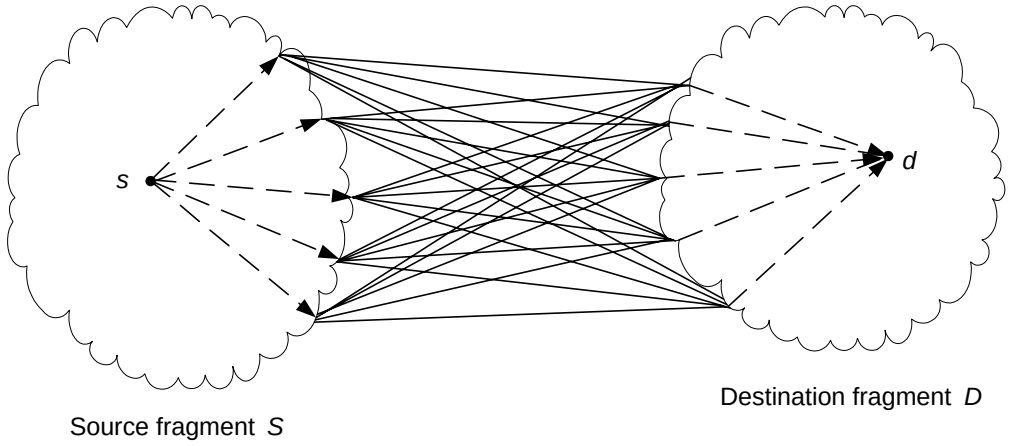

Figure 2.2: Exhaustive Comparing Algorithm

In Figure 2.2, the dashed lines in S and D represent the shortest paths from s to boundary vertices in S or from boundary vertices in D to d. The solid lines represent the shortest paths from boundary vertices in S to the boundary vertices in D. The shortest paths within S and D are easy to acquire, just applying Dijkstra's algorithm on the fragments. To find the shortest distance between any u and v, one must go up one level and see if they are in the same fragment in the higher level. If so, that means the shortest distance is already materialized and it only need to be read from the routing table on disk. If not, going up one level further to see if they are in the same fragment or not is necessary, until the top level is reached, where every pair of boundary vertices has it shortest distance materialized. Assume that there are k levels in the hierarchy; the worst case is that for every query of $SD(u, v)$ in $(*)$ we have to go to the top level (level k). Assume that the number of boundary vertices in each fragment is the same, say b. In the ground level (level 1), we need to submit b^2 shortest distance queries to level 2. For every such query,

level 2 also needs to submit b^2 shortest distance queries to level 3, and so on, until the top level is reached. Therefore, there are totally $(b^2)^{k-1}$ routing table lookups on the top level. Usually it can be assumed that $b = O(\sqrt(n))$, so the complexity of exhaustive comparing algorithm on multilevel hierarchical architecture is $O(n^{k-1}$. When k is larger than 3, the asymptotic complexity of the exhaustive comparing algorithm is worse than Dijkstra's algorithm, except that it can cope with larger graphs. Therefore, if a disk-based Dijkstra's algorithm is possible, it should run faster.

2.2.3 Rooted Tree Method

The rooted tree method [41, 42] is also a disk-based shortest path algorithm based on divide-and-conquer. Its partitioning algorithm is an external-memory extension of the Lipton and Tarjan's planar separator algorithm. Therefore, it can only partition planar graphs. Also the disk-based data structure selected for storing pre-computation information requires a lot of space, thus may not suitable for very large digital maps. To see why is this case, the examination of what is stored in the rooted tree and how the shortest path algorithm works is necessary.

Pre-processing Phase: Constructing Rooted Tree

The rooted tree data structure is a d-nary disk-based tree structure. Each node of the tree is associated with a connected graph and its planar graph separator. The children of a node are the connected components separated by its separator, i.e. let G be a parent, S be the separator of G, $G_1, G_2, \ldots, G_k$ be the resulting partitions

of G, if G_i has more than one vertex, they should be have their own separtors S_i, otherwise the vertex itself is the separator. Then the parent of the the $\bigcup G_i = G \backslash S$. This is illustrated in the Figure 2.3

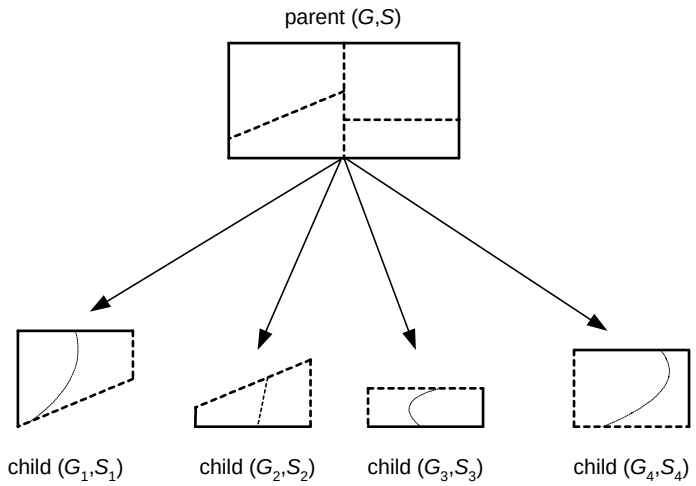

Figure 2.3: Rooted Tree (dotted lines are separators)

The digital graph is first partitioned into two connected components A and B using planar separator C. If A or B is not small enough, it is partitioned recursively. In a certain stage, the partitioning algorithm pauses and all the connected components generated from the last pause form one level of the rooted tree. For example, in Figure 2.3, the parent graph G was first divided into two connected components, then the two components are divided further into four connected components. These four components (G_1, G_2, G_3 and G_4) together with their separators (S_1, S_2, S_3, and S_4) form the next level of (G, S) in the rooted tree. If G_i is not small enough, they are divided further as shown by the finer dotted lines. Note that the children of a graph G do not include vertices of the separator. Therefore,

this guarantees that every vertex appears in exactly one node in the rooted tree if the tree satisfies the two conditions:

1. At each level, only separators are stored.

2. Each leaf node contains only one vertex.

Given a vertex v, we can find exactly one node (G_k, S_k) in the rooted tree such that v is in S_k (i.e. v is a boundary vertex of G_k), and v is an interior vertex of its ancestors $G_i, (i < k)$. Given two vertices u and v, we can find their lowest level common ancestor (G_l, S_l). Define $B(u, v)$ to be the union of the all boundary vertices in their common ancestor, i.e. $B(u, v) = \bigcup S_i, (i \leq l)$. It can be proved that the shortest path between u and v must pass through at least one vertex in $B(u, v)$. Intuitively, this is correct since if at least one of u or v is in S_l (one vertex is another's ancestor), the claim is obviously true. If otherwise u and v are not in S_l, then u and v must be the interior vertices of G_l. The vertices of $B(u, v) - S_l$ are the boundary vertices of connected components that encompass the connected component G_l. Therefore, the shortest path must pass through either some vertex in S_l or some vertex in $B(u, v) - S_l$.

Based on this observation, the shortest distance from s to d can be written as : $dist(s, d) = \min(dist(s, b) + dist(b, d))$, where b is the boundary vertices in the ancestor of x, $dist$ is the local shortest distance between two vertices. The shortest distance query then can be implemented by an exhaustive comparative algorithm: for each boundary vertex b in the $B(s, d)$, find the minimum value of $dist(s, b) + dist(b, d)$. If the shortest distance from any interior vertex to any boundary vertex is already known for each node in the rooted tree, the shortest

distance query problem is boiled down to how to efficiently retrieve the shortest distance from one interior vertex to a boundary vertex. The thesis [42] discussed the external memory data structure for storing the rooted tree for efficient I/O.

The problem in the above method is that it requires a lot of disk space for storing the shortest distance for the rooted tree and a lot of pre-computation for the shortest distances. Assume that the ground level digital map has n vertices. Therefore, there are $\sqrt{n}$ boundary vertices on the root if using the Lipton and Tarjan's planar graph separator. For each interior vertex, there should be a shortest distance to every boundary vertex stored in the rooted tree data structure. That is there are $\Theta(n^{3/2})$ shortest distance computations for the pre-computation and $\Theta(n^{3/2})$ shortest distances stored in the tree. This will be too much computation and the storage requirement will be too great. For example, in California, there are about 1 million vertices. Then there will be 1 billion shortest path computations and 1G shortest distances just for the root of the tree.

2.3 Our New Disk-based Shortest Path Algorithm

In order to get around the problems and limitations of the previous methods, it is proposed that some practical improvements and new algorithms targeting on very large digital maps be developed. The general idea is the same as described in section 2.2.1, but we use different partitioning algorithms, hierarchical scheme, pre-computation materialization scheme, and shortest path querying algorithms. Our contributions lie in the following aspects:

- A partitioning algorithm based on BFS and Hilbert R-Tree is issued. The

advantages of this algorithm are that it is scalable and can be easily extended to certain semantics.

- A 2-level hierarchical graph instead of multi-level (≥ 3) hierarchies are used. The more the levels in the hierarchy, the more work needed for answering shortest path queries. 2-level hierarchical may have a upper bound for the size of the digital map, but it can handle a reasonable large digital map (for example, in the scale of 10^8 vertices and edges) on contemporary personal computers .

- I/O is optimized by clustering the results of pre-computation, and storing them as objects in the spatial database.

- The results of pre-computation are stored in a disk-based data structure – a virtual hash table, rather than routing tables or rooted tree.

- An auxiliary data structure "sketch graph" is used to capture the outline of the super graph and to help to prune the super graph when answering the shortest path queries.

- A different disk-based shortest path algorithm similar to Dijkstra's algorithm is used, rather than exhaustive comparative algorithm used by the routing table approach and rooted tree method.

Chapter 3

Design of the Shortest Path Query Engine

3.1 Problems Trying to Solve

Based on the discussion in last chapter, we have a clear idea on how the general framework works. However, there are still many practical difficulties when designing the concrete algorithms. In particular, the following questions must be answered:

1. How to partition the digital map if the whole graph cannot be seen? It is required that the partitioning algorithm must work correctly even if it only has a partial image of the whole digital map at any time. This will be answered in Section 3.2.

2. What information should be recorded in the pre-computation? How fast can it be done? And what data structure should be used in order to retrieve it

efficiently in the query-answering phase? Keeping the size of storage as small as possible to save the storage space without the loss of any information is a goal. These questions are answered in Section 3.3 and 3.4.

3. Is there any way to prune the searching space without the loss of optimality of the shortest path? What additional information should be kept in the pre-computation phase? The pruning process is particularly useful when the source and destination vertices are close to each other while the searching space is huge. This question is answered in Section 3.4 and 3.5.

4. What the effect of different data structures on the disk-based shortest path algorithm? What are the criteria of choosing a particular data structure? This question will be discussed in Section 3.6 and later in Chapter 5 on the performance analysis of different data structures.

3.2 Graph Partitioning Algorithm

The input of our graph partitioning algorithm is a general graph, and may or may not be planar, so Lipton and Tarjan's planar separator is not a good candidate. The output is a set of fragments (or rimless fragments) as defined in Section 1.2. Throughout the discussion of this partitioning algorithm, it is assumed that only part of the graph can be loaded into main memory at any time, while the rest of the graph is available on disk.

3.2.1 Algorithm Description

No matter what the graph is (planar or non-planar) or where it is stored, the graph must somehow be traversed in order to partition it. A good candidate of a graph traversing algorithm is the breadth-first search (BFS), which is also used in the Lipton and Tarjan's planar separator algorithm. The difference from their approach is that the bread-first spanning tree is not constructed for the whole graph in our algorithm. Rather the algorithm frequently pause traversing when a certain condition is satisfied, then the traversed vertices and edges are extracted to form a fragment and saved to the disk. Then the fragment is removed from main memory to save space for other untraversed part of the graph to load in. This heuristic algorithm cannot guarantee the optimality of the vertex separator (i.e. no more than $2\sqrt{2n}$ boundary vertices) as done by the Lipton and Tarjan's algorithm, but the payoff is its simplicity to implement and its capability to partition non-planar graphs. Another nice property of this algorithm is that it can be easily extended to accommodate particular semantics. For example, the digital map can be partitioned such that some special type of street blocks, say interstate highway, can only be on the boundary of fragments. In this case, the pause condition is that all the next boundary vertices ready for BFS exploring are vertices corresponding to interstate highway intersections. A more realistic case is to combine many conditions together to form a pause function. Whenever the function returns true, the exploring process pauses and saves the result.

To get around the problem raised by question 1 in Section 3.1, it is necessary to have the capability to read in any region of the digital map. This can be done by

indexing the geometric objects in the digital map by means of a Hilbert R-tree. By Hilbert R-tree, geometric objects (line features in our algorithm) are clustered in Minimum Bounded Retangles (MBR), which is represented by the coordinates of the right-upper corner and the left-lower corner of the rectangle). Given any MBR, we can retrieve all geometric objects that intersect the rectangle. Then a region of the digital map can be constructed by assembling all the returned geometric objects into a graph. Regions can be merged together to form a "window" (or view) of the digital map, in which we can do BFS exploring without touching any vertices or edges outside the window. For example, the grids in Figure 3.1 represent the regions in a digital map. The shaded regions constitute a window, of which grid 0 is the center.

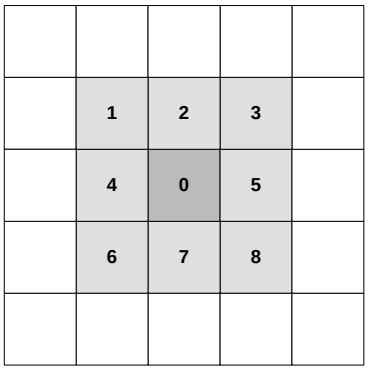

Figure 3.1: Grids in a Digital Map

If the following two requirements can be satisfied:

1. The window can grow to cover all the regions of the digital map eventually.
2. In any time, the BFS process only explores vertices and edges within this window.

Then it can be easily proved that our BFS searching based on window moving explores all the vertices and edges in the digital map, since at each step of BFS in the window, the visible adjacent vertices and incident edges of a vertex are the same as those in a BFS on the whole graph in main memory.

The algorithm of preparing and maintaining the window for BFS exploring is called "Prepare", which is shown in Algorithm 1. The goal of the algorithm is to guarantee all the neighbor vertices and incident edges of a vertex v is inside the window. Therefore, when we explore on it, BFS would not loss any adjacency information of that vertex.

In this algorithm, the graph G represents the window. After preparing, G is guaranteed to contain all the adjacent vertices and incident edges of v. However there is a special case. If there is an edge outside the window so long that the other end vertex, say u, is not in any of the eight neighbor regions of the center region, the region containing u will not be merged into G by the Prepare algorithm. Notice that the edge (v, u) must intersect one of the eight neighbor regions. The edge must have been merged into G after preparing, so the post-condition still hold. Later, when u is the next vertex to be prepared, the region containing it and all the neighbor regions thereof will be merged into G (line 8-18 of the algorithm).

The size of G is non-decreasing in terms of the preparing process, but after a fragment (which should be a subgraph of G) is stored on disk, it can be removed from G. Therefore, the window would not grow as large as the digital map. The partitioning algorithm is shown in Algorithm 2.

This algorithm is based on the Hilbert R-Tree and breadth-first search (BSF).

Algorithm 1 PREPARE(G,v,C) {Prepare for BFS}

Input: the graph G which will be prepared for BFS, the vertex v which is the next vertex to explore by BFS, the current central region C.
Output: well prepared G for v
Require: the digital map is already divided into grid regions
Ensure: all v's the incident arcs and adjacent vertices are merged into G

```
1: if v is in MBR(C) then {if any of the 8 neighbor regions of C has not been
   merged into G, merge it}
2:    for all neighbor C' of C do
3:       if C' has not been merged into G then
4:          G ⇐ G + C'' {merge C'' into G}
5:       end if
6:    end for
7: else {v is not in the region MBR(C)}
8:    C ⇐ region(v) {find the region in which v is}
9:    G ⇐ G + C {merge C'' into G}
10:   for all neighbor C' of C do {if any of the 8 neighbor regions of C' has not
      been merged into G, merge it}
11:      if v is in MBR(C') then
12:         for all neighbor C'' of C' do
13:            if C'' has not been merged in G then
14:               G ⇐ G + C'' {merge C'' into G}
15:            end if
16:         end for
17:      end if
18:   end for
19: end if
20: set the color of newly added vertices in G to be WHITE
```

Algorithm 2 PARTITION(Map,min_size,max_size) {Partitioning the digital map}

Input: Map is a digital map indexed by Hilbert R-tree, min_size and max_size are the prefered minimum and maximum size of fragments respectively.
Output: a fragment database F
Require: $1 < min\_size \leq max\_size < 2 * min\_size$.

1: Divide Map into grid regions.
2: Initialize C, v, and G {randomly pickup a region C, v in C, $G \leftarrow \emptyset$}
3: PREPARE(G, v, C) {prepare G for BFS from v}
4: $Q \leftarrow V(G)$ {initialize queue Q to contain all vertices in G}
5: **while** $Q \neq \emptyset$ **do**
6: **if** f.size $\geq min\_size$ **then** {time to pause}
7: $F = F \cup \{f\}$ {add fragment f to database}
8: $G \leftarrow G - f$ {remove f from "window" G}
9: $Q \leftarrow$ RandomlyPickFrom(Q) {clear all but one vertices in Q}
10: **else**
11: $u \leftarrow Q$.dequeue()
12: **for all** $v \in$ Adjacent(u) **do**
13: $f = f + v + (u, v)$ {add vertex v and edge (u, v) to fragment f}
14: **if** v.COLOR = WHITE **then**
15: v.COLOR $\leftarrow$ GRAY
16: **end if**
17: **end for**
18: u.COLOR = BLACK {u can be safely removed from main memory now}
19: **end if**
20: **end while**
21: **if** $\exists v$ such that v.COLOR = GRAY **then**
22: Q.enqueue(v)
23: goto 5
24: **end if**
25: **if** $\exists v$ such that v.COLOR = WHITE **then**
26: v.COLOR = GRAY
27: Q.enqueue(v)
28: goto 5
29: **end if**
30: Merge tiny fragments into larger ones if possible
31: **for all** $f_1 \in F$ **do**
32: **for all** $f_2 \in F \wedge f_2 \neq f_1$ **do**
33: Find the boundary set between f_1 and f_2.
34: **end for**
35: **end for**

It takes three parameters:

- A geometric database (digital map) indexed by Hilbert R-Tree.
- The minimum number (lower bound) of vertices in the resulting fragments.
- The maximum number (upper bound) of vertices in the resulting fragments.

The output is a fragment database containing fragments covering the digital map. The size (number of vertices) of the fragments should be less than maximum number.

In this algorithm, the digital map is first divided into small enough regions, each of which should be smaller than the minimum size of fragment. Then we randomly pick one region as the central region C and one vertex v in C as root to start BFS exploring. Since an edge could cross the border of two adjacent regions when we do a breadth-first-search, we have to "prepare" the graph for the central region and the root vertex before exploring the graph. Each vertex in the graph has a color property, which can be one of the three colors: WHITE, GRAY or BLACK. Before exploring the graph, all vertices are initially "painted" to be WHITE. When a vertex is seen the first time, its color is set to be GRAY, indicating that it is already "discovered", but it is not known whether its adjacent vertices, if any, have been discovered or not. BLACK vertices are those vertices that all of its neighbors are discovered (i.e. all of its neighbors are of color GRAY or BLACK). By giving each vertex a color property, we can divide the vertices into three sets, which require different treatments. All GRAY vertices of the current fragment are stored in a queue Q, and every time we want to go on exploring the graph, we should

get the next vertex from Q, i.e. a GRAY vertex. Thus the GRAY vertices are the border vertices of the current fragment. (Note that a border vertex is not the same as a boundary vertex. It is a candidate of a boundary vertex. A gray vertex may not be a boundary vertex as explained later in this section). The BFS algorithm enlarges the fragment by keeping picking a border vertex and adding its adjacent vertices to Q until some condition comes true, in which case a fragment is generated and can be written to the database. When all neighbors of a GRAY vertex have been discovered, the GRAY vertex can be darkened to BLACK, indicating that it is an interior vertex of the current fragment. Since an interior vertex has no edges connecting to interior vertices in the other fragment, it is safe to remove interior vertices and all incident edges from the graph without affecting other fragments. This will save memory space for other vertices and edges loading into memory. When Q is empty, we still need to check if GRAY vertices or WHITE vertices exist as shown in line 21-28. GRAY vertices may exist when Q is empty because Q only carries the GRAY vertices of the current fragment. It is possible that half of the graph has been explored and the north bound of the map is reached, therefore Q is empty. But the source half of the graph is still not explored, and the GRAY vertices are right there waiting.

The reason for checking the existence of WHITE vertices when Q is empty and there are no GRAY vertices, is that the digital map may not be connected. If starting from one connected component, we cannot reach the other connected components by BFS. Therefore, we have to check out the existence of WHITE vertices for other connected components. Different connected components belong

to different fragments even if they are smaller than the minimum size.

3.2.2 Proof of Correctness

By definition, a partition is a set of fragments where the union of the vertices in fragments equals to the set of vertices in the original graph. First it is proven that the sub-graphs generated by algorithm Graph-Partitioning are rimless fragments. Then it is proven that every vertex in the original graph is in some fragment of the partitioning, and every vertex in each fragment is in the original graph.

Now it is shown that when the pause condition (line 6 in Algorithm 2) is satisfied, the sub-graph written out is actually a rimless fragment. By definition in section 1.2, a rimless fragment is a special kind of sub-graph such that the vertices can be divided into two sets: interior vertices and boundary vertices, where interior vertices cannot be an empty set. Actually, when the pause condition is satisfied, the BLACK vertices in current fragment are a subset of interior vertices, and the GRAY vertices are a super set of boundary vertices. Since $min\_size > 1$ by precondition, the root vertex and all of its adjacent vertices are always added into the current fragment if they are not already in the current fragment. The edges connecting the root vertex to its adjacent vertices are added to the current fragment as well. This means that the degree of the BLACK vertex in the current fragment is the same as the degree of the corresponding vertex in the original graph. Thus the BLACK vertices are a subset of interior vertices. By definition, a boundary vertex connects interior vertices in different fragments. When the stoppage condition is satisfied, the vertices in Q are all GRAY, and they are the only GRAY vertices in current

fragment. A GRAY vertex in Q occurs in two cases:

1. All of its adjacent vertices are interior vertices in the current fragment.

2. An adjacent vertex exists, but not in the current fragment.

In the first case, the GRAY vertex is actually an interior vertex of the current fragment since it is not adjacent to any interior vertices in other fragments. It will be merged into the current fragment in line 30 of the algorithm. In the second case, the GRAY vertex is a real boundary vertex since its degree in the current fragment is less than its degree in the original graph. Thus the GRAY vertices are a superset of boundary vertices. From the description above, it is known that for the current fragment, at least one BLACK vertex and its adjacent vertices in current fragment can be found, thus the interior vertices set is not empty; or the whole fragment is merged into another fragment. Therefore, the sub-graph we get is actually a fragment.

It is obvious that all the vertices in fragments are in the original graph by the characteristics of BFS graph traversal algorithms. Now it will be shown that every vertex in the original graph is in some fragment(s).

The original graph can be either connected or unconnected. For connected graphs, if $|V| \leq min\_size$, the stoppage condition in line 6 will never be satisfied before Q gets empty. Therefore, the whole graph is a fragment and there is no boundary vertex. If $|V| > min\_size$, the condition 6 will be eventually satisfied before Q gets empty. In this case, the current fragment (all BLACK vertices, GRAY vertices, and all edges connecting them) is saved to a spatial database. A GRAY vertex is picked up as the new root for BFS either from Q or from a GRAY node

repository as in line 21-29 in the algorithm. The code between line 21 to 29 is necessary since BLACK vertices in the current fragment are removed from the original graph. The remaining graph could be still connected or becoming unconnected. In either case, there must be at least one vertex in each connected component that is of the color GRAY, otherwise it is contradictory to the assumption that the original graph is connected. The existence of the GRAY vertex repository guarantees all connected components are explored. Thus no WHITE vertex exists when the algorithm terminates, which means every vertex must be saved in some fragment.

If the original graph is not connected, then we start off from one connected component and the algorithm guarantees that all the vertices in this connected component are stored in some fragment. Line 21-29 of the algorithm will check if there are still WHITE vertices, which means there exists other connected components that are not explored. If so, the algorithm will pick one WHITE vertex in the connected component and do the same procedure until no WHITE vertices are left. This guarantees that all vertices are BLACK or GRAY and should be saved in some fragment in the database. Thus the resulting fragments are in fact a partition of the original graph.

To sum up, it has been proven that all vertices in the original graph are in some fragment(s) and all vertices in the fragments are in the original graph. Furthermore, for each fragment, the BLACK vertices are interior vertices, the GRAY vertices are boundary vertices

3.2.3 Complexity Analysis

Since the algorithm does a lot of I/O operations, we have to take into account both CPU usage and I/O costs. The I/O cost is measured by the number of geometric objects that are read in or written out.

In terms of I/O cost, since caching is used in persistent data structures (see Section 3.2.6), the actual I/O cost depends on the buffer size and the query pattern, which is not known at this time. In order to get around of this, only the worst case is considered. That is, there is only one entry in the cache buffer, and every time a data entry is accessed, it must be from the disk. In the initialization step (Line 1 and 2 of the algorithm), the I/O costs are $2m$, where m is the number of edges in the digital map. From line 5 to 29, the whole digital map is traversed once and the resulting fragments are stored in the spatial database. Therefore, the I/O costs are also $2m$. In line 30, if there are no assistant data structures, each pair of fragments and the common vertices between them should be checked for. Since the upper bound of fragment size is max_size, and there are at most m/min_size fragments of size larger or equal to min_size, the upper bound of I/O costs is $(1+2+3+\ldots+ m/min\_size)*max\_size = (m+min\_size)*\mathrm{m}*max\_size/min\_size$. Since the actual max_size is usually less than $2min\_size$, the upper bound is $2m(min\_size + m)$. Since m is usually huge, this operation is very costly. In practice, if an MBR for each fragment is kept in main memory, it reduces the I/O cost drastically. If this is being done, for each fragment, we only read those fragments that have an overloading MBR. Since the number of such fragments is usually 3 to 5, the actual I/O cost for line 30 is around $5m$. If using a caching technique, the I/O cost could

be reduced even more depending on the number of buffers. Similarly for line 31 to 34, if no extra data structure is used, the I/O cost is in $O(m^2)$. However, if the MBR's of fragments are checked before accessing the fragment, the I/O cost is around $5m$ also. Therefore, the total I/O cost is around $2m+2m+5m+5m = 14m$.

In terms of CPU time, step 1 to 4 take time in order $O(m)$, since they are all linear operations on edges of the digital map. In step 5 and 6, the whole digital map is traversed by BFS algorithm, so the time complexity is $O(n + m)$. In step 7 and 8, if MBR is used, each vertex in one fragment is looked up in its adjacent fragments. Therefore, the worst case is in $O(n \log max\_size)$, assuming that looking up vertex in a fragment takes $O(\log max\_size)$ time, where max_size is the maximum fragment size. We will see in chapter 4, that the looking-up operation in fact takes $O(\log max\_size)$ time. Hence the upper bound of CPU running time of the graph partitioning algorithm is $O(n \log(max\_size) + m)$.

3.2.4 Adaptability to Particular Semantics

A good thing about this algorithm is that you can easily extend it to accommodate different semantics by replacing the queue data structure Q with a priority queue, and applying a pause checking function in line 6 of Algorithm 2 rather than just comparing the size of current fragment with the minimum size. Vertices are given priorities according to the preference of being interior vertices. Therefore, when we get the next vertex from Q to explore, we will always get the vertex with the lowest priority which prefer being a boundary vertex. A pause checking function can help to determine where to stop according to the lowest priority in Q, as

well as the size of the current fragment. This is a more flexible algorithm than the SPC partitioning algorithm in [10]. Also it does not bias the near square or circle digital maps, as SPC does. It is scalable since the input data are stored in geometric object files and the output fragments are stored in persistent data structures. The quality of the partitioning algorithm is also very good. Because of the properties of BFS, the fragment naturally resembles a circle or square in itself, and the number of boundary vertices can also be minimized by such heuristics that assign the higher degree vertices higher priorities. In this way, the boundary vertices of one fragment by and large connect to less (boundary) vertices of other fragments. Thus the total number of boundary vertices and the number of edges connecting those boundary vertices should be minimized. Experiments show that the average number of adjacent fragments is around 3 to 5, with almost half of the number are 4. This is comparative to the grid meshing method.

3.3 k-pair shortest paths algorithm

After the digital map is partitioning into fragments, each fragment has zero or more boundary sets depending on whether it shares vertices with adjacent fragments or not. If the number of boundary sets is non-zero, boundary vertices are push up one level to form the next higher level of super graph. In addition, there should be a super edge connecting each pair of boundary vertices. Therefore, it is necessary to computer the shortest paths between every pair of boundary vertices in the fragment.

The easiest way is to use the all-pair shortest paths algorithm directly to the

fragment. When the all-pair shortest paths among all vertices in a fragment are found, it is trivial to get the all-pair shortest paths among boundary vertices. There are two main methods for computing all-pair shortest paths: iterative Dijkstra's algorithm and dynamic programming. The iterative Dijkstra's algorithm is straightforward: simply apply Dijkstra's shortest path algorithm on each vertex. The Johnson's algorithm introduced in [1] is based on this idea. This approach is effective when the graph is sparse. The complexity is $O(mn + n^2 \log n)$, where m is the number of edges, n is the number of vertices.

Dynamic programming is aother approach, which construct the all-pair shortest paths in a bottom-up way so that the latter shortest paths are built upon previous shortest paths. The complexity of this algorithm is $O(n^3)$. This algorithm is effective when the graph is dense, since then $m = \Theta(n^2)$ and the Johnson's algorithm is also in $O(n^3)$. In this case, the dynamic programming approach is more efficient than Johnson's algorithm since there is a small constant due to its simplicity. The dynamic programming approach can be even faster if the base shortest paths are chosen carefully and the shortest paths are ordered in a Fibonacci heap. In [25], the authors introduced a new algorithm based on dynamic programming that runs in $O(m^*n + n^2 \log n)$, where m^* is the number of edges in the all-pair shortest paths. This algorithm is likely to be fast in practice because it is already known with high probability in many distributions of the edge weights [25].

Although dynamic programming could be more efficient than iterative Dijkstra's algorithm in the case of all-pair shortest paths, it usually is not the case in k-pair shortest paths ($k \ll n$). In our application, the number of boundary vertices is

in the order of $\sqrt{n}$, where n is the number of vertices in a fragment. Computing all-pair shortest paths in a fragment will incur a lot of overhead. For example, if the size of a fragment is around 10,000, the average number of boundary vertices is about 300, then about $(\frac{30}{10,000})^2 \approx 0.1\%$ of CPU time is actually spent on the result we want. Therefore, our k-pair shortest path algorithm is based on iterative Dijkstra's shortest path algorithm with some heuristics to speed up the process.

The heuristics we added to the iterative Dijkstra's algorithm is based on the observation that the boundary vertices are not uniformly distributed in the fragment, but clustered as boundary sets. Since many boundary vertices are adjacent to each other, the shortest path from one boundary vertex to another boundary vertex is very likely to pass through other boundary vertices. This will save many runs of Dijkstra's algorithm because if the shortest path from u to v passes through another vertex w, then it is obvious that the shortest paths from u to w and w to v are overlapping with the shortest path from u to v, otherwise the shortest path from u to v is not the shortest. Moreover, if there are more than two vertices, say $w_1, w_2, \ldots, w_k$, lie in the shortest path from u to v, the shortest paths between u to w_1, w_i to w_{i+1} (where 1 $\leq$i$\leq$ $k-1$), and w_k to v are also known. This is shown in the Algorithm 3 line 9-14.

Our algorithm first creates a matrix M, where $M_{i,j}$ is the shortest distance from boundary vertex v_i to v_j. At first, every element in matrix M is initialized to $+\infty$. Whenever a shortest distance between v_i and v_j is found, it is filled out in $M_{i,j}$. Then we check whether there are other boundary vertices on the path. If so, assign the correct value to the corresponding element in the matrix. The idea of this

Algorithm 3 k-Pair-SP(f, B) {k-pair Shortest Path Algorithm}

Input: f is a fragment, B is the set of boundary vertices in f

Output: a $|B| \times |B|$ matrix M containing the shortest distances between every pair of vertices in B

Require: $B \subseteq V(f)$

Ensure: $M[i,j]$ contains the shortest distance from vertex i to j, $M[i,j] = +\infty$ if no path from i to j

```
1: if i = j then {initialize the matrix}
2:    M[i,j] = 0
3: else
4:    M[i,j] = +∞
5: end if
6: for all u ∈ B do
7:    for all v ∈ B ∧ v ≠ u do
8:       if M[u,v] = +∞ then
9:          p ← SP(f,u,v) {find the shortest path from u to v in fragment f}
10:         Find {w_1, w_2, ..., w_k} ⊆ B ∩ V(p), where u = w_1, v = w_k
11:         for 1 ≤ i ≤ k do
12:            for i < j ≤ k do
13:               M[w_i, w_j] ← dist(w_j) − dist(w_i)
14:            end for
15:         end for
16:      end if
17:   end for
18: end for
```

heuristic is to reuse the information calculated for one run of Dijkstra's algorithm as much as possible. Although only sequential algorithms are considered in this thesis, this idea is particular suitable for parallel processing in which each thread is dedicated to one run of Dijkstra's algorithm on one pair of boundary vertices. All threads share one distance matrix. One thread can make use of the results of other threads and also can contribute to the matrix.

3.4 Sketch Graph

In the approach of [10], nothing is done to the super graph except that the super graphs are partitioned further to generate higher-level graphs. Since the performance deteriorates greatly when the hierarchical graph gets too many levels (more than three), it is not an appropriate method for very large digital maps such as the whole road system of the United States. In our algorithm, the digital maps, and the ground level graphs, are partitioned and the super graphs are stored in a spatial database (virtual hash table). There are only two levels in the hierarchy. The benefits of having fewer levels in hierarchical graph is that the performance can be guaranteed for a large map. The downside is that the super graph is also huge when the digital map is at the size of more than 100,000 vertices. Finding the shortest paths on the huge super graph may cost hours using the conventional graph traversal algorithms since it requires a lot of I/O.

To tackle the difficulties, a data structure called a "sketch graph" was created which captures the high-level outline of the super graph and is much smaller. Moreover, the more interesting thing about sketch graph is that it can be used for pruning

the super graph when given source and destination vertices. A pruning algorithm – vertical pruning (or α-β pruning) – can be applied to the sketch graph. First, the sketch graph and the algorithm generating the sketch graph are introduced in this section. Then the vertical pruning algorithm is introduced in the next section.

According to the definition in section 1.2, a super graph contains only boundary vertices in the ground level graph, and for pairs of boundary vertices in the same fragment, there is an edge connecting them. Therefore, it is actually a very dense graph that consists of cliques. The number of cliques is the number of fragments in the ground level graph. The property of a graph that consists of cliques is that the number of boundary vertices is not that large, but the number of edges is. For example, for the road systems in Connecticut, the number of vertices in the ground level graph is about 160,000, and the number of edges is about 190,000. When using the graph partitioning algorithm introduced in section 3.2.1, the number of boundary vertices is about 1200 for 15 fragments. The size of the boundary vertices in each fragment varies from 120 to 290. The total number of edges in the super graph is about 190,000, approximately the same size of the ground level graph. Finding the shortest paths in the super graph is of the same complexity as finding them in the ground level graph, and requires a similar amount of main memory. Thus divide-and-conquer does not help much. When looking at the super graph carefully, some nice properties can be discovered that allow you not take into consideration the whole super graph yet still get the optimal solution.

One of these nice properties is that the boundary vertices in each fragment are fully connected, and they can be partitioned by equivalence sets called boundary

sets as defined in section 1.2. Two boundary vertices are in the same equivalence set if, and only if, they have the same adjacent vertices in the super graph, i.e. vertices in the same boundary set have the same connectivity properties. Therefore, a boundary set can be safely contracted into a super vertex without losing its connectivity properties. If two boundary sets are fully connected (they form cliques), we can connect such two vertices with an edge. It can be easily proven that the reversion of the contraction process (expanding a vertex in the sketch graph to a set of vertices and connecting each pair of vertices in the two sets if and only if the two vertices in the sketch graph are connected) will restore the connectivity in the original super graph.

In addition to the connectivity property, the sketch graph also keeps the minimum and maximum distances from vertices in one boundary set to the vertices in another boundary set (α-value and β-value). In this way, part of the shortest distance information is reserved in the sketch graph, and it can be used to prune the sketch graph later. The algorithm for generating the sketch graph is as shown in Algorithm 4.

The sketch graph consists of cliques, but in a very small size, since the number of vertices (boundary sets in original fragments) in the clique is small (usually 3 5), the average degree of vertices is about 6 to 10, so $|E|$ is around $3 * |V|$ to $5 * |V|$, which is very sparse compared to the super graph. In the same example of road systems in Connecticut state, the original graph is divided into 15 fragments. The number of vertices and number of edges in the sketch graph are 21 and 65 respectively. In such a small and sparse graph, the vertical pruning algorithm based on Dijkstra's

Algorithm 4 Get-Sketch(g_s) {Get the sketch graph from super graph}

Input: the super graph g_s
Output: the sketch graph k

```
1: for all fragment f in in g_s do
2:    for all boundary set b in f do
3:       construct a super vertex v corresponding to b
4:       if v is not in k then
5:          k ← k + v
6:       end if
7:    end for
8:    for all boundary set x in f do
9:       for all boundary set y in f and x ≠ y do
10:         e ← (x, y) {create a super edge e}
11:         e.α ← min(SP(x_i, y_i)), ∀x_i ∈ x, y_i ∈ y
12:         e.β ← SP(x*, y*) {x* and y* are the delegate vertices of boundary sets
            x and y respectively}
13:         k = k + e {merge super edge e to sketch graph}
14:      end for
15:   end for
16: end for
```

algorithm (in $O(n \log n + m)$) can be done almost instantly. We hope that the extra efforts of pruning are worth the time saved on exploring the super graph. In Chapter 5, we will see the experimental results to see if it is true in the real world.

3.5 Pruning Algorithm

If the sketch graph is large and the source and destination vertices are very close, intuitively it is not necessary to explore the whole graph to find the shortest path. The sketch graph can be pruned in such a way that even if some super nodes and super arcs are eliminated, the optimal solution can still be found.

3.5.1 Algorithm Description

In order to get the lossless pruning, it is necessary to keep more information in the sketch graph – α and β values – associated with super edges in the sketch graph. According to the definition of sketch graph, each edge associates a 2-tuple (α, β), where α and β respectively are the minimum and maximum values of the shortest distance from any boundary vertex in one boundary set, to any boundary vertex in another boundary set. Knowing this fact, a range in which the shortest distance from one vertex to where the other vertex falls could be found. The vertical pruning algorithm is used to calculate this range and delete any vertices and incident edges that should not be passed through. Otherwise the sum of the shortest distance from source to this vertex and from this vertex to the destination exceeds the upper bound. This algorithm can also be called α-β pruning. It takes three parameters: the sketch graph, source, and destination vertices. The result is

a pruned sketch graph. The algorithm is shown in Algorithm 5.

Algorithm 5 Pruning(k,s,d,S,D) {Pruning Sketch Graph}

Input: k is the sketch graph, s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are in respectively.
Output: a subgraph of k
Require: s is in S, d is in D
Ensure: the shortest path from s to d remain the same on k and the pruned sketch graph
1: add vertices s and d to k
2: add edges connecting s to the boundary sets in S and edges connecting d to the boundary sets in D
3: label the edges of sketch with their β values
4: find the shortest distance $SD_\beta(s,d)$ from s to d in k
5: label the edges of sketch with their α values
6: select s as root and perform shortest path algorithm to get the shortest path tree T_s
7: select d as root and perform shortest path algorithm to get the shortest path tree T_d
8: **for all** vertex v in k **do**
9: **if** $T_s(s,v) + T_d(v,d) > SD_\beta(s,d)$ **then**
10: $k \leftarrow k - v$ {remove the super vertex v and its incident super edges from k}
11: **end if**
12: **end for**

In line 3 of the algorithm, the sketch graph is converted to a general graph (called β-graph) with the edges being labeled by β. In line 4, the shortest distance $SD_\beta(s,d)$ obtained from β-graph is actually the upper bound for the shortest distance from s to d. In line 5-7, the sketch graph is converted to a general graph (called α-graph) by labeling the edges with their α values. The shortest path trees from s and d are computed in the a-graph. Since we are considering the undirected graph only, the shortest path tree from d to other vertices is actually the shortest path tree from other vertices to d. For the directed graph, we can reverse

the direction of each edge first, and then apply the single source shortest path tree algorithm. The complexities remain the same.

3.5.2 Proof of Correctness

In order to prove that the correctness of the vertical pruning algorithm, we have to justify two statements:

1. The shortest distance from s to d in the β-graph $SD_\beta(s,d)$ is an upper bound for the shortest distance $SD(s,d)$.

2. If there exists a boundary set B in the super graph such that $\alpha(s,B) + \alpha(B,d) > SD_\beta(s,d)$, the shortest path cannot pass through any vertex in B. Therefore, it is safe to remove B and all the incident edges.

Proof of the first statement:

It must be proven to be true in two cases:

1. If s and d are in the same fragment, $SD_\beta(s,d) = SD(s,u) + SD(v,d)$, where u and v are boundary vertices in some boundary set such that $SD(s,u) \geq SD(s,v)$ according to the definition of β-graph. Therefore, we get $SD_\beta(s,d) \geq SD(s,v) + SD(v,d) \geq SD(s,d)$. Thus $SD_\beta(s,d)$ is an upper bound for $SD(s,d)$.

2. If s and d are in different fragments, and we assume that the shortest path in β-graph passes n boundary sets ($n > 0$). Then $SD_\beta(s,d) = SD(s,u_1) + SD(v_1,u_2) + SD(v_2,u_3) + \ldots + SD(v_i,u_{i+1}) + \ldots + SD(v_{n-1},u_n) + SD(v_n,d)$,

where u_i, v_i are boundary vertices in the i^{th} boundary set. By definition of β-graph, $SD_\beta(s,d) \leq SD(s,v_1)+SD(v_1,v_2)+SD(v_2,v_3)+\ldots+SD(v_i,v_{i+1})+\ldots+SD(v_{n-1},v_n)+SD(v_n,d) \leq SD(s,d)$. Thus $SD_\beta(s,d)$ is an upper bound for $SD(s,d)$.

Proof of the second statement:

Prove by contradiction: if the shortest path passes through a vertex v in boundary set B and $\alpha(s,B)+\alpha(B,d) > SD_\beta(s,d)$. Since we know that $SD(s,v) \geq \alpha(s,B)$ and $SD(v,d) \geq \alpha(B,d)$, $SD(s,d) = SD(s,v)+SD(v,d) \geq \alpha(s,B)+\alpha(B,d) > SDb(s,d)$. We know that $SD_\beta(s,d)$ is the upper bound of $SD(s,d)$, so $SD(s,d) \leq SDb(s,d)$. There is a contradiction. Thus the shortest path cannot pass through a vertex in boundary set B.

3.5.3 Complexity Analysis

The running time can also be divided into two parts: I/O time and CPU time. I/O only takes place in step one. If a dictionary for the MBR's in each fragment is not available, it is necessary to read through all fragments in the spatial database in the worst case. Therefore, it takes m reads, where m is the number of geometric objects in the spatial database. However, if the MBR's of all fragments are kept in main memory, to determine which fragment contains s or d, one needs only to read constant fragments from the spatial database. Therefore, the I/O costs depend only on the size of fragments.

For CPU time, step 2 can be done in $O(n_f \log n_f + m_f)$, where n_f is the number of vertices in the fragment, and m_f is the number of edges in the fragment. In step

3 and 5, we have to traverse every edge in the sketch graph, so the time complexity is $O(m_s)$, where m_s is the number of edges in the sketch graph. In step 4, 6 and 7, we use Dijkstra's shortest path algorithm and the complexity is $O(n_s \log n_s + m_s)$, where n_s is the number of vertices in the sketch graph. In step 8, we check each vertices in the sketch graph, so the time complexity is $O(n_s)$. Therefore, the overall time complexity is $O(n_f \log n_f + m_f + n_s \log n_s + m_s)$.

3.6 A Disk-based Shortest Path Algorithm

After pruning a sketch graph, we can construct a super graph by reading all boundary vertices and super edges from the super graph database, and merging the fragments containing source and destination vertices with the super graph, then the shortest path can be found by applying Dijkstra's algorithm on the resulting graph. This is feasible for small digital maps, but not for very large digital maps, since the size of the super graph is almost the same as or even larger than the digital map, so the same problem, insufficient main memory, also persists. In this case a disk-based algorithm needs to be designed to store part of the information on hard disk, and load it dynamically when needed. The differences between the main memory shortest path algorithms and disk-based shortest path algorithm are discussed in the following section.

3.6.1 Differences from the Previous Algorithms

My disk-based algorithm is also based on the idea of Dijkstra's algorithm, i.e. keeping track of the shortest distance information in a data structure, organizing

the vertices in an ascending order on the current shortest distance from the source, keeping "closing" vertices with the minimum shortest distance so far, until no vertex is open. The main differences between the disk-based shortest path algorithm and the previous algorithms rely on the following two aspects:

1. Whether to use external memory data structures or not.

2. Whether to use pre-calculated information or not.

For the main memory Dijkstra's algorithm, the information of vertices (current shortest distance from the source, the preceding vertex in the current shortest path, and whether the vertex is closed or not) is stored in a hash table or tree. Retrieving this information is very fast due to fast memory access and little computation. In disk-based algorithm, since the information is too large to fit into memory, part of it should be stored on disk. Whenever the information needs to be accessed, an I/O operation is necessary. Therefore, one of the goals of the new data structures is to minimize the number of I/O operations. With main memory algorithms, the vertices could be ordered in a binary heap or Fibonacci heap. In [19], Goldberg and Tarjan showed that a binary heap could be more efficient than a Fibonacci heap in a sparse graph due to relatively small number of $decreaseKey$ operations, although a Fibonacci heap has smaller asymptotic order. Recently, new data structures were developed for special input graphs. For example, in [20, 21, 22], Cherkassky, Goldberg, and Silverstein tested the performances using different data structures (buckets, multilevel buckets, hot-spot queues, heap-on-top queues) on organizing vertices according to their distances. Some very interesting results came out for special graphs such as the weights of edges are bounded in a range of integers.

However, none of them are designed to facilitate disk-based algorithms, i.e. their performances rely on the fact that data can be randomly accessed in main memory in constant time. Our new data structure has to consider the situation in which the main memory is limited and data needs to be swapped out to disk. The relationship between the previous main memory data structure used in Dijkstra's algorithm and our data structure is like the relationship between the binary search tree and the B-tree.

The second difference is whether to use the materialized information that is computed before the query phase to answer the query. All the main memory algorithms mentioned above do not make use of the result of pre-computation. Whenever a new shortest path query comes, it just starts all over again. In contrast, the pre-computed information is materialized (stored) to hard disk in disk-based algorithm. It can be used over and over again for new queries. For example, in this algorithm, the fragment database generated by the partitioning algorithm, the shortest distance matrix generated by the k-pair shortest paths algorithm, and the pruned sketch graph generated by the pruning algorithm are reused. This can save a lot of work. However, if the underlying digital map is not static, for example, the weight of the edge can be changed, and changing the materialized pre-computation database needs a lot of time. Therefore, the query result may not be up-to-date until the materialized information has been updated. This is a trade-off between optimality and efficiency.

3.6.2 Algorithm Description

The disk-based shortest path algorithm takes seven inputs: the source node s and the destination node d, the two fragments S and D in which s and d are contained respectively, the whole set of fragments (also called fragment database) $frags$, the set of distance matrices (also called distance database) $matrixDB$ one for each fragment, and the pruned sketch graph sketch. The data structure for holding the fragments and distance matrices are disk-based data structure – virtual hash tables. The detailed discussion of virtual data structure will be defered to Chapter 4. At the time being, it is necessary to know that a virtual hash table can be treated as its main memory counterpart – hash table. That is, to store an object, the object as well as a key to that object are required. To retrieve an object, only given a key of that object is necessary.

Intuitively, we can get the shortest path by merging the source and destination fragments S and D with the super graph, and then apply Dijkstra's shortest path algorithm on the merged graph. The difficulty in this is that the super graph is too large to fit into main memory. My approach is to make the super graph a disk-based data structure, in which a set of buffers (a.k.a cache) in main memory is maintained. Part of the super graph can be loaded from disk when needed and flush some entries to disk when buffer is full and new entries are required.

Initially all vertices in S, D and all boundary vertices are open and their distance from s is infinity except s itself which is 0. The algorithm first constructs a shortest path tree from s to every vertex in fragment S. This process can be done using Dijkstra's algorithm since the fragment is small enough to fit into main memory.

During the process, whenever a boundary vertex is closed, the relaxation process in Dijkstra's algorithm [1] should be applied to any adjacent boundary vertex in the super graph. Since boundary vertices for super graph are stored on disk and the distances from the closed boundary vertex to its adjacent boundary vertices are also stored on disk, this process may incur I/O operations if they are not in cache yet. In each iteration, the vertex with the minimum shortest distance so far is closed. The relaxation and closing process keeps going until all boundary vertices of fragment D are closed. The next step is to find the shortest distance from every boundary vertex in D to destination vertex d. After this we have get a "guideline" of the shortest path. The guideline consists of three parts: the shortest path from s to the last boundary vertex b_s in S, the shortest path from the first boundary vertex b_d in D to d, and a sequence of boundary vertices on the shortest path between b_s and b_d. The first two parts are a complete comparison to the resulting shortest path, but the third part may have missing interior vertices between each pair of boundary vertices. The last step of this algorithm is to "fill up" the missing vertices by looking up the intermediate fragments from the fragment database and apply Dijkstra's algorithm to these fragments.

In order to minimize the I/O operations, it is preferable to cluster the boundary vertices in such a way that adjacent boundary vertices are in as few clusters as possible. In our specific application, probably the most efficient way is to cluster the boundary vertices is fragment clustering. That is, two boundary vertices are in the same cluster if, and only if, they are in the same fragment. In this way, a relaxation only need to read 2 to 4 clusters (since usually one boundary vertex is

in no more than 4 fragments). However, since a boundary vertex is in more than one fragment, then you have to load at least two fragments, say F_1 and F_2, for updating one fragment, and these two fragments have many common boundary vertices - those in the boundary set . The boundary set is read twice, thus incurs I/O overhead. Another way to cluster boundary vertices is by boundary sets. Since very few boundary vertices are in more than one boundary set, it has very little I/O overhead comparing to fragment clustering. The downside of boundary set clustering is that boundary sets must be read one by one, rather than reading several boundary sets in the same fragment in batch. However, since a boundary set is usually large (in the scale of 50 vertices), it does not loose the advantages of buffered reading and writing. Therefore, boundary set clustering is a better way than fragment clustering in general.

As with Dijkstra's shortest path algorithm, we keep track of information for each vertex, which includes the shortest distance from s so far, the parent node in the shortest path, and whether it is closed or not. In Dijkstra's algorithm, nodes are stored in a priority queue, usually binary heap, ordered by their distance from s. In my algorithm, nodes are organized in three kinds of priority queues (heaps) as shown in Figure 3.2.

Nodes in S and D can be put in a binary heap $interQ$ as in Dijkstra's algorithm. However, unlike Dijkstra's algorithm, the minimum value a in $interQ$ is not necessarily the next closed node. Rather it should be compared with the minimum value of boundary vertices b, and the smaller one of these two is the next closed vertex. Boundary vertices are clustered into boundary sets. The attributes of a boundary

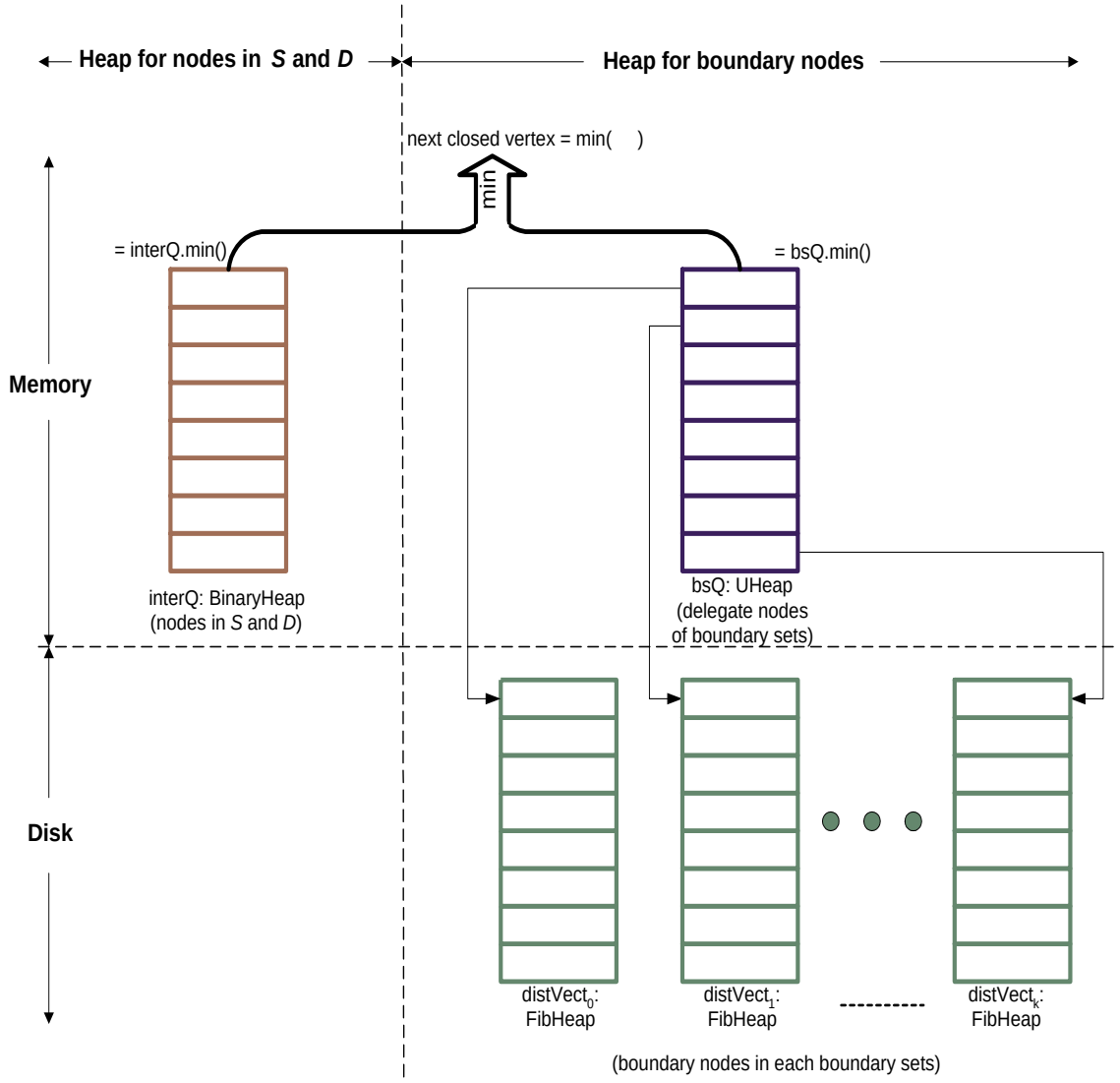

Figure 3.2: Data Structures for Disk-based Shortest Path Algorithm

set are stored in a data structure, Distance Vector ($distVect$), which is stored on disk. The boundary vertices in a distance vector are organized in a Fibonacci heap ordered by their distance from s. Each boundary set has a delegate boundary ver-

tex which is put into a global "Updatable Heap (U-Heap)" (see section 3.6.3 for detail) in main memory. The selection of the delegate is based on the criteria that its shortest distance from s is the minimum within all open boundary vertices in the boundary set. The Updatable Heap is similar to the Mergable Heap introduced in [5], but with an additional method that allows updating the key of an entry in the heap. The algorithm is sketched in Algorithm 6.

The algorithm works like this: in step 1 we initialize the two queues $interQ$ and bsQ for nodes in S and D, and delegate nodes in boundary sets respectively. For each boundary set, a distance vector is also constructed (e.g. an internal Fibonacci heap is generated) to hold all boundary vertices in this boundary set. These distance vectors are stored in a virtual hash table $dvDB$ with the boundary set ID as the key. Initially all vertices in S and D expect s itself have a distance of ∞ from s. Distances from s to every boundary vertex is also set to be ∞ in the distance vector initially. In step 2 of the algorithm, we initialize the distance of s to be 0 and the closed property to be true. Step 3 is the main part of this algorithm. It is same as the relaxation part of Dijkstra's algorithm, except that there is an additional step to do the main thrust if the next closed vertex is a boundary vertex. Main thrust is shown in Algorithm 7 and the discussion will be delayed until later in this section. After step 3, every vertex in S and every boundary vertex is closed, but interior vertices in D are not closed, if S and D are not equal. Therefore, in step 4, we push all boundary vertices in D into the binary heap $interQ$ (which should be empty after step 3), and find the shortest path from these boundary vertices to d. Step 4 is the same as Dijkstra's algorithm if we draw an edge between s and every

Algorithm 6 DiskSP(s,d,S,D,F,M,k)

Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are in respectively, F is the fragment database, M is the distance matrix database, k is the sketch graph

Output: the shortest path from s to d

Require: s is in S, d is in D

1: **for all** vertex $v \in S$ **do** {initialize UHeap and distance vectors for boundary sets}
2: v.distance= $+\infty$
3: $interQ$.enqueue(v)
4: **end for**
5: Initialize distance vector database
6: bsQ.enqueue(dv.delegate())
7: s.distance $\leftarrow 0$ {initialize binary heap for interior vertices}
8: s.closed = TRUE
9: $interQ$.enqueue(s)
10: $g \leftarrow S + D$ {merge S and D to graph g}
11: **while** $\neg bsQ$.empty() $\vee \neg interQ$.empty() **do**
12: $a \leftarrow interQ$.min()
13: $b \leftarrow bsQ$.min()
14: **if** $interQ$.empty() $\vee$ (b.distance ¡ a.distance) **then**
15: do MainThrust on b {relax all boundary vertices adjacent to b}
16: b.closed = TRUE
17: **else**
18: $interQ$.dequeue()
19: relax all vertices adjacent to a in g
20: **if** a is boundary vertex **then**
21: do MainThrust on a
22: **end if**
23: a.closed = TRUE
24: **end if**
25: **end while**
26: **for all** vertex $v \in D$ **do** {find the shortest paths from boundary vertices to d within fragment D}
27: $interQ$.enqueue(v)
28: **end for**
29: **while** $\neg interQ$.empty() **do**
30: $a = interQ$.dequeueMin()
31: relax all vertices adjacent to a in g
32: a.closed = TRUE
33: **end while**
34: FillSP() {construct the complete shortest path from the simplified shortest path got from steps 1 to 4}

boundary vertex in D with the edge weight being the shortest distances between them so far as seen from step 3. After step 4, we get an incomplete shortest path, which can be seen as concatenation of three parts: a shortest path from s to the last boundary vertex b_s in S (by the last vertex we mean that all its subsequent vertices in the shortest path are not in S while its previous vertex in the shortest path is in S), a simplified shortest path from b_s to the first boundary vertex b_d in D (by first we mean all the subsequent vertices in the shortest path are in D while the previous vertex in the shortest path is not in D), and a shortest path from b_d to d in D. It is shown in Figure 3.3.

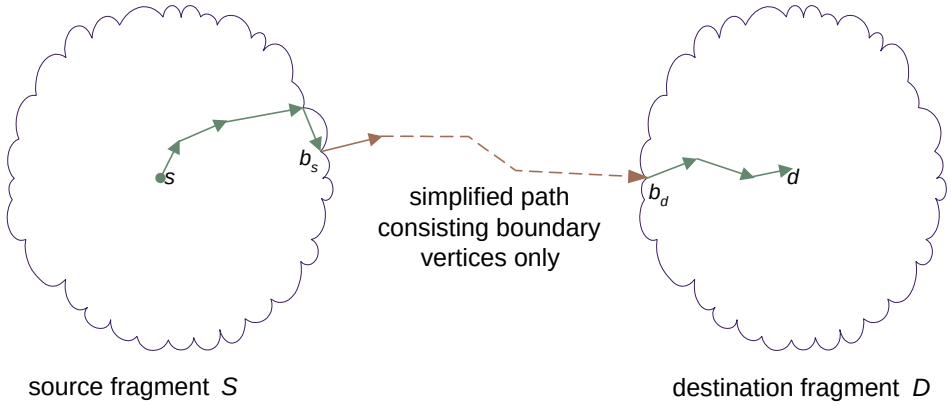

Figure 3.3: Simplified Shortest Path

In step 5, we complete the simplified path (dashed line in Figure 3.8) by looking at the fragments through which it passes. This algorithm is shown in Figure 3.10.

First we see what main thrust does in Figure 3.9. The purpose of main thrust is to relax all adjacent boundary vertices of a boundary vertex in the super graph. First in step 1, we find the boundary set in which the boundary vertex u is. Since a boundary set is all boundary vertices shared by two fragments (F_1 and F_2), all

Algorithm 7 MainThrust(u,d,k,bsQ,$dvDB$,M)

Input: u is the next closed boundary vertex, d is the distance of u, k is the sketch graph, bsQ is the U-Heap containing delegates of boundary sets, $dvDB$ is the distance vector of boundary sets, M is the distance matrix

Output: none

Ensure: every boundary vertex adjacent to u is relaxed

```
1: find the boundary set bs = [F1, F2] in which u is in
2: find the adjacent boundary sets B of bs from k
3: separate boundary sets in B into two lists L1 and L2 such that ∀b1 ∈ L1, b1 ⊆
   F1, ∀b2 ∈ L2, b2 ⊆ F2, and L1 ∪ L2 = B
4: for all boundary set b ∈ L1 do
5:    m = M.get(F1) {get the distance matrix for fragment F1 from database}
6:    for all boundary vertex v ∈ b do {do relaxation}
7:       if v.distance > d + m.get(u,v) then
8:          v.distance = d + m.get(u,v)
9:          v.predecessor = u
10:      end if
11:   end for
12:   bsQ.updateValue(b,b.distance) {keep bsQ's heap property}
13: end for
14: for all boundary set b ∈ L2 do
15:   m = M.get(F2) {get the distance matrix for fragment F2 from database}
16:   for all boundary vertex v ∈ b do {do relaxation}
17:      if v.distance > d + m.get(u,v) then
18:         v.distance = d + m.get(u,v)
19:         v.predecessor = u
20:      end if
21:   end for
22:   bsQ.updateValue(b,b.distance) {keep bsQ's heap property}
23: end for
```

the boundary vertices in F_1 and F_2 are adjacent to u in the super graph and should be relaxed. In the $DiskSP$ algorithm, there are two places that call $MainThrust$. The first is in line 4 of step 3. Here b is the delegate boundary vertex of a boundary set. Therefore, the boundary set is already known and finding which one is b is necessary. The second place is in line 10 of step 3, where the boundary vertex in the source fragment S is the next closed vertex. If the boundary vertex is in only one boundary set, we can just iterate through all the boundary sets and find it. Otherwise, if the boundary vertex is in multiple boundary sets, we can arbitrarily choose one and let it be the boundary set in which b is and do $MainThrust$. It will be proven later that this does not affect the correctness of the shortest path algorithm. In step 2, all the boundary vertices in F_1 and F_2 are found by looking up their boundary sets from sketch graph. In the sketch graph, the nodes represent the boundary sets in the fragments. There is an edge between two nodes if the two corresponding boundary sets are in the same fragment. Therefore, finding all boundary sets in F_1 and F_2 can be done by finding all adjacent nodes in the sketch graph. In step 3, we divide the boundary sets into two sets according to which fragment they belong to (bs is in both fragments so it should be in both sets). The reason for this is that we have to retrieve the fragment's distance matrix from a virtual data structure. Grouping boundary sets by fragment can reduce the number of retrievals to the virtual data structure, thus reducing I/O cost when the buffer of virtual data structure is small. Step 4 and 5 are doing the same thing - relaxation on every boundary vertex – in L_1 and L_2 respectively.

Now the $FillSP$ algorithm is introduced which is shown in Algorithm 8. The

correctness proof of $DiskSP$ is deferred to section 3.6.4 after the data structure is introduced.

Algorithm 8 FillSP(F,$dvDB$,S,D,s,d)

Input: F is the fragment database, $dvDB$ is the distance vector database, S and D are the source and destination fragments respectively, s and d are the source and destination vertices respectively.

Output: a complete path from s to d

Require: the shortest path tree has been established, i.e. every vertex has the shortest distance and predecessor records

Ensure: the path returned is the compatible with the shortest path tree, i.e. the shortest path

1: $p \leftarrow d$
2: **while** $p \neq s \wedge \neg p$.isBoundary() **do** {construct part within fragment D}
3: $pre = p$.predecessor
4: $nodes$.push(pre) {push vertex pre to the nodes stack}
5: $edges$.push((pre, p)) {push edge (pre, p) to the edges stack}
6: $p \leftarrow pre$
7: **end while**
8: $pre \leftarrow p$.predecessor
9: **while** $p \notin S \wedge p$ is not the last boundary vertex in the shortest path **do** {construct the part between fragments D and S}
10: find the shortest path $subp$ between pre and p
11: $nodes$.push($V(subp)$) {push all vertices in $subp$ into nodes stack}
12: $edges$.push($E(subp)$) {push all vertices in $subp$ into nodes stack}
13: $pre \leftarrow p$.predecessor
14: **end while**
15: **while** $p \neq s$ **do** {construct the part within fragment S}
16: $pre \leftarrow p$.predecessor
17: $nodes$.push(pre) {push vertex pre to the nodes stack}
18: $edges$.push((pre, p)) {push edge (pre, p) to the edges stack}
19: $p \leftarrow pre$
20: **end while**
21: $path \leftarrow$ CONSTRUCT-PATH($nodes$,$edges$)

The purpose of the $FillSP$ algorithm is to complete the simplified shortest path by filling out the missing interior vertices in the interior fragments. The completion

is done backward from vertex d in D to vertex s in S. The first step is to find the vertices and edges of the shortest path within the fragment D and put them into two stacks respectively. The second step finds the actual shortest paths between two boundary vertices in the interior fragments. This is done by applying Dijkstra's algorithm on the interior fragments. The vertices and edges of the shortest paths are also put in the same stacks. The third step is to find the vertices and edges with the shortest path with the fragment S. Again the vertices and edges are put into the stacks. The last step is to construct the shortest path from the node stack and edge stack.

3.6.3 Data structures

In this section the following topics are introduced: the updatable heap (U-Heap), distance vector and distance matrix data structures used in disk-based shortest path algorithm, and why we choose these data structures (binary heap, U-Heap, Fibonacci heap) in the algorithm.

A U-Heap implements a priority queue interface with the major two operations:

- FindMin(): Returns the object with the minimum value in the heap.
- UpdateValue(key): Update the value of an object in the heap according to the given key.

An example of an updatable table is shown in Figure 3.4.

An updatable heap can be thought of as a full and complete binary tree, in which only the leaf nodes contain data. A binary tree is a full binary tree if each

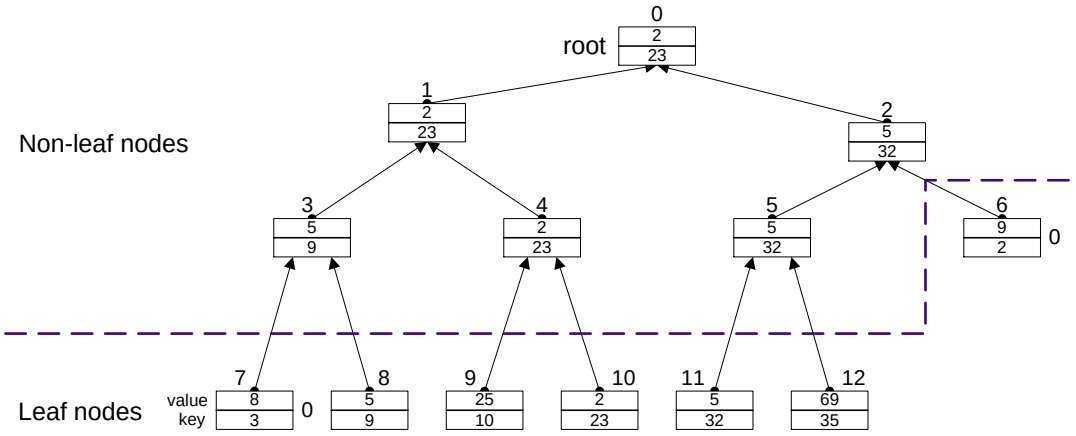

Figure 3.4: U-Heap Data Structure

node has zero children or exactly two children. A complete binary tree is a binary tree in which all leaf nodes are at level h or $h-1$, where h is the height of the tree, and all nodes in level h are toward left [47]. It can be easily proved by mathematical induction that given any number of data items, a full and complete binary tree can always be constructed, such that all the data items are leaf nodes of the tree. The total number of nodes in a U-Heap is $2n-1$, where n is the number of data items (leaf nodes) in the U-Heap. A U-Heap can be implemented using an array, in which the parent of the i^{th} element $a[i]$ is $a[(i-1)/2]$, and its two children are $a[2i+1]$ and $a[2i+2]$.

In a U-Heap, each node is composed of two fields: key and value. The key field is any comparable data structure and can be duplicated. The value field is any object associated with the key. They have different usage in the U-Heap data structure. At first, the key field is used to sort the leaf nodes in the binary tree. Once they are sorted, their positions are fixed at the bottom level. The value field

is then used to construct the tree structure bottom up. That is, to compare the values of each pair of nodes which belong to the same parent, so that the node with the minimum value are pushed up one level as the parent. This process keeps going until the root of the tree is generated. Therefore, a non-leaf node has the minimum value among all of the leaf nodes in its subtree. Particularly, the root node contains the minimum valued leaf node. Therefore, the running time of $FindMin$ operation is $\Theta(1)$. The $UpdateValue$ operation consists of two steps: finding the leaf nodes which have the given key, and updating all the ancestors of these satisfied leaf nodes. By sorting the leaf nodes by their keys, it is easy to find a leaf node in $\Theta(\log n)$ given the key. Since the height of the tree is in $\Theta(\log n)$, updating the ancestors of a given leaf node also requires $\Theta(\log n)$ time, so the total asymptotic complexity of UpdateValue operation is $\Theta(\log n)$.

A distance matrix is a very simple data structure that keeps the static information about the shortest distance of every pair of vertices in a fragment. Its purpose is to provide a fast way to answer the shortest distance query between two vertices in the fragment. There is only one major method in its interface: $get(c_1, c_2)$, where c_1 and c_2 are the coordinates of two vertices. It returns the shortest distance between these two vertices within this fragment. The shortest distances can be stored as a nxn matrix M, where n is the number of vertices in the fragment. Then the shortest distance from the i^{th} vertex to the j^{th} vertex is given by $M[i, j]$. The problem for this is that we have to define a mapping from the coordinate of a vertex to its index on the matrix. This can be done by storing the coordinate and index in a hash table, or a sorted array. When the original graph is undirected, the path

from the i^{th} vertex to the j^{th} vertex is the same as the path from the j^{th} vertex to the i^{th} vertex. Therefore, we can delete half of the matrix (upper triangle or lower triangle) to save the space. That is, only $M[i,j]$, $i < j$ is saved (all $M[i,i] = 0$).

A distance vector is a data structure of boundary sets to keep track of the shortest distance information from the source vertex. In the disk-based algorithm, when a boundary vertex is chosen to be the next closed vertex, all of its adjacent boundary vertices in the super graph should be relaxed. Since all boundary vertices are clustered into boundary set, a batch relaxation for a boundary set can be done. In order to do the batched relaxation, the following information about the boundary set is necessary.

- A collection of open vertices, so that the necessary vertices will be updated.
- A way to keep the precedent vertex of a boundary vertex in the shortest path.
- The minimum value among all shortest distances from source vertex to the open vertices after relaxation so that the delegate vertex for this boundary set can be updated.

To provide the first and section information, the open vertices are organized in a hash table keyed by their coordinate and keep the open/closed information for each boundary vertex, as well as the precedent vertex. To provide the third information quickly, it is more difficult. A priority queue is used to organize the open vertices. Various kinds of heaps are candidates for this data structure; for example binary heaps, Fibonacci heaps, and et al. Binary heaps are practically more efficient when the graph is sparse, i.e. when there are few vertices that need

to decrease their keys. When the key of one vertex is determined to decrease, we can only insert the vertex again with the new key and save a lot of time from eliminating the expensive decreaseKey operations. Since inserting a vertex multiple times may make the heap overflow, we have to dynamically expand the heap space when needed. If there very few vertices that need to decrease their keys, the expansions seldom happen, so the average running time is lower. However, if there are many vertices that need to decrease keys, the binary heap is not as efficient since eventually the cost of expansion will exceed the cost of $decreaseKey$ operations, in which case the Fibonacci heap probably is the most suitable data structure. In our specific application, the super graph is composed of cliques, each corresponds to one fragment. When one boundary vertex is to be closed, all boundary vertices in the fragment should be relaxed. This may incur a lot of $decreaseKey$ operations because the graph is dense.

As shown in Figure 3.6, three kinds of heaps are used for the three types of priority queues. This is due to the usage pattern of these three priority queues. The $interQ$ priority queue is used for organizing the vertices in sparse graphs, fragment S and D. Since there are few $decreaseKey$ operations in sparse graphs, binary heap is the most suitable data structure for $interQ$. On the other hand, $distVect$ contains the boundary vertices in a boundary set, which is a subset of the super graph. Since the super graph is a dense graph composed of complete graphs, Fibonacci heap is more suitable for $distVect$. Unlike $interQ$ and $distVect$, bsQ is a priority queue that contains delegate vertices for boundary sets. Since the number of $updateValue$ operations is large (the worst case is one $updateValue$ operation

for each boundary vertex), using binary heap is not appropriate because it does not provide an efficient way to search a particular node in the heap. Also because updating a value of a node in the priority queue is not always decreasing the value, Fibonacci heaps are not appropriate for this situation either. Therefore, U-Heap is the most appropriate data structure for bsQ among these three data structures.

3.6.4 Correctness Proof

To prove that the $DiskSP$ algorithm is correct, it is only necessary to prove that:

1. The $DiskSP$ is equivalent to the Dijkstra's algorithm on the graph $S + D + SuperGraph$, where "+" is the graph merging operator and $SuperGraph$ is a properly pruned super graph of the digital map. Since the graph-pruning algorithm is independent to the $DiskSP$ algorithm, it has already been proved correct in section 3.5.2. Without the loss of generality, we assume that $SuperGraph$ is the unpruned super graph.

2. The result found from step 1) is the same as the result found from applying Dijkstra's algorithm directly on the digital map.

In Step 1, the super graph is composed of cliques, one for each fragment. The vertices in a clique are all the boundary vertices in a fragment. The edge weights are the shortest distances from its start vertex to its end vertex. In this algorithm the edge weights of a fragment are stored in a $DistMatrix$. It will be shown that the combination of a sketch graph and all of the $DistMatrix$'s can replace the need of super graph in the Dijkstra's algorithm. In Dijkstra's algorithm on $S + D + SuperGraph$, all the open vertices (including those in S, D and

$SuperGraph$) are stored in a priority queue. The vertex with the minimum shortest distance is extracted from the queue to be the next closed vertex. In the $DiskSP$ algorithm, vertices are split into two parts: vertices in S and D are stored in a priority queue ($interQ$) as in the Dijkstra's algorithm; boundary vertices are stored on disk, clustered by boundary sets. One vertex in each boundary set with the minimum shortest distance is selected and put in another priority queue (bsQ) in the main memory. The minimum between the two minimum values in $interQ$ and bsQ is extracted to be the next closed vertex. Therefore, the shortest distance of the next closed vertex is the minimum among all open vertices in $S + D + SuperGraph$. Therefore, the Dijkstra's algorithm on $S + D + Supergraph$ and the $DiskSP$ are the same if the set of open vertices and their shortest distance information is the same.

Then it will be shown that after each open vertex is closed, the relaxation processes in both Dijkstra's and $DiskSP$ algorithm give the same shortest path information for all vertices in $S + D + SuperGraph$. At first, the initial step is the same, the source vertex s is selected as the next closed vertex and its shortest distance is 0. Then all vertices adjacent to s are relaxed in Dijkstra's algorithm. Assume that s is not a boundary vertex, then in $DiskSP$ algorithm, the relaxations happens on the same set of vertices. Suppose that at some step, a boundary vertex v is chosen to be the next closed vertex, all the adjacent vertices to v should be relaxed in the Dijkstra's algorithm. These vertices include the interior vertices, if any, in $S + D$ and boundary vertices $SuperGraph$. Note that a boundary vertex in boundary set $[i, j]$ is adjacent to every boundary vertex in fragment i and j.

Therefore, every boundary vertex in fragment i and j should be relaxed. This is exactly what the $MainThrust$ procedure in the $DiskSP$ algorithm does. Note that a boundary vertex in the super graph may be in more than one boundary set, that is, it may be in more than two fragments. This could be true in our partitioning algorithm. It will be shown that even in this situation, the $MainThrust$ operation can guarantee that all copies of the boundary vertices indifferent to boundary sets are consistent, i.e. their shortest distance from s equals to the optimal one. At the time being, it is assumed that all boundary vertices are in exactly one boundary set, so a boundary vertex is relaxed in $DiskSP$ algorithm if, and only if, it is relaxed in the Dijkstra's algorithm, and the relaxation update the same set of boundary vertices with the same information. Therefore, at each step, a boundary vertex is closed and the same set of vertices are updated with the same information. By induction, the results of Dijkstra's algorithm on $S + D + SuperGraph$ and $DiskSP$ algorithm are the same.

Next it is proven that even if a boundary vertex is contained in more than one boundary set, the $MainThrust$ procedure still gives the same result. First, an example of why a boundary vertex can be contained in more than two fragments. In Figure 3.5, suppose that u, v, w, x, and y are open vertices and we start partitioning from fragment F_1. When u is closed (all its adjacent vertices including v and w are explored) a fragment stop point is reached and all the vertices and edges explored are saved to form a fragment. Therefore, F_1 contains u, v and w. Then w is chosen to be next root for BFS to generate another fragment. After x is closed (v is explored again), another stop point is reached and F_2 is generated. Therefore,

F_2 contains v and w.

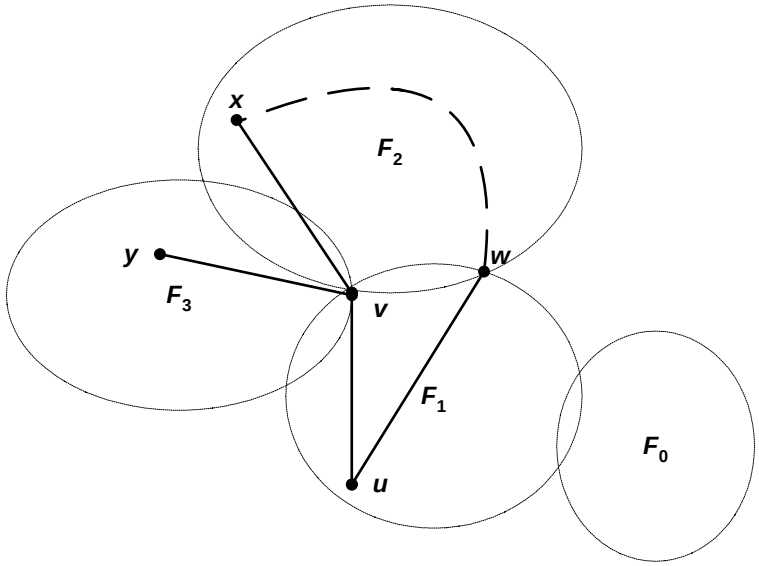

Figure 3.5: A partitioning on the graph containing vertices u, v, w, x and y: solid lines represent edges, dashed line connecting w and x represent a path between them, small dashed ellipses represent fragments F_0, F_1, F_2, and F_3

Suppose we select v as the next root for BFS, y is eventually explored and should be contained in the third fragment, say F_3. (Note: x could be a boundary vertex shared by F_2 and F_3 depending on whether x is adjacent to an interior vertex in F_3 or not. For simplicity, it is assumed that x is not in F_3). In this case, vertex v is contained in three fragment and three boundary sets - $[F_1, F_2]$, $[F_1, F_3]$ and $[F_2, F_3]$. The difficulty for v in more than one boundary set is that the information (shortest distance and so on) are stored in each of these boundary sets, and they may be inconsistent at some point in time. However, it will be shown that once the first copy of v is to be closed, all the other copies are consistent to it and their shortest distances could not be less than the first closed copy. The idea behind this is

that a boundary vertex can be reached from different boundary vertices in different fragments. Their shortest distance from s could be different. However, once it is determined that the shortest distance from one fragment should be the "real" shortest distance, all the shortest distances information in other copies should be set to this value. Therefore, the inconsistences of multiple copies before the closing of one copy does not affect the correctness as long as they are consistent after one copy is closed.

For example in Figure 3.12, assume the source fragment $S = F_0$. Whenever a boundary vertex in the boundary set $[F_0, F_1]$ is closed, all boundary vertices in $[F_1, F_2]$ and $[F_1, F_3]$ should be relaxed. Therefore, the information of v stored in $[F_1, F_2]$ and $[F_1, F_3]$ are updated and should be the same. However, v in $[F_2, F_3]$ may not be the same as those in $[F_1, F_2]$ and $[F_1, F_3]$, but once one of the copies of v in these boundary sets is closed, the shortest distance information for all other copies in different boundary sets should be the same. No matter which copy of v in $[F_1, F_2]$, $[F_1, F_3]$ or $[F_2, F_3]$ is the next closed vertex, it should be the minimum value among all copies. Without the loss of generality, assume that v in $[F_1, F_2]$ is the first closed in these three boundary sets, then all the boundary vertices in F_1 and F_2 should be relaxed, thus the copies of v in $[F_1, F_3]$ and $[F_2, F_3]$ should be relaxed and their shortest distance values should be the same value as v in $[F_1, F_2]$ (since the shortest distance from v to itself is always 0). After the relaxation, the distance information for all copies of v in different boundary sets are consistent, and v's in $[F_2, F_3]$ and $[F_1, F_3]$ should be the next closed vertices if no other vertices are equal to the same shortest distance value. It will been

proven that the result of $DiskSP$ algorithm is the same as the result of Dijkstra's algorithm on $S + D + SuperGraph$. That the result of Dijkstra's algorithm on $S + D + SuperGraph$ is the same as the result of Dijkstra's algorithm on the digital map still needs to be proven. In fact, one instance of Dijkstra's algorithm can be represented by a sequence of vertices associated with the shortest distances from s. For example, the sequence $p =< s(0), u)3), v(5), w(5), x(8), d(10) >$ represents an instance of Dijkstra's algorithm on a graph containing vertices s, u, v, w, x and d. The order is the closed order of these vertices, which also defines the index of a vertex in the sequence. For example vertex v can be represented by $p[2]$. The number in the parentheses is the shortest distance from s to the vertex at the point it is closed, and also is the "real" shortest distance. The shortest distance of a vertex v in the sequence p is denoted by $SD_p(v)$, or $SD_p[2]$ if the index is used. Also the subsequence between u and v in sequence p is denoted by $p(u, v)$. The prefix of sequence p before u (exclusive) is denoted by $p(-, u)$. Similarly, the suffix of a sequence p after u (exclusive) is denoted by $p(u, -)$. First, two definitions used in the proof are offered. We define that two such sequences are equivalent if, and only if:

1. the two sets of vertices are the same, and

2. the shortest distances associated with each vertex are the same, and

3. the order of the vertices are the same unless they have the same shortest distance value.

For example, the shortest distance of v and w are the same in the above example, so it is equivalent to the sequence $< s(0), u(3), w(5), v(5), x(8), d(10) >$ but not others.

We define that one shortest path sequence p is compatible with another shortest path sequence q if, and only if:

1. the vertices in p is a subset of the vertices in q, with a restriction that the first and last vertices in q must appear in p, and

2. the shortest distance associated with each of the vertices in p is the same as the shortest distance associated with corresponding vertices in q, and

3. the total order of vertices in p is a partial order of the corresponding vertices in some equivalent subsequent of q.

It is easy to see that if two sequences are equivalent, they get the same shortest distance from s to d. If one sequence is compatible with another sequence, they will also get the same shortest distance from s to d. Therefore, if the results of instances of an algorithm are always compatible with the results of instances of another algorithm, it can be said that the two algorithms always get the same results. That the sequences generated by Dijkstra's algorithm on $S + D + SuperGraph$ are always compatible with the sequences generated by Dijkstra's algorithm on the digital map directly will be shown next.

To prove that the sequence generated by the Dijkstra's algorithm on $S + D + SuperGraph$ (can be thought of as a Turing Machine TM_1) s_1 is compatible with the sequence generated by the Dijkstra's algorithm on the digital map (suppose the

Turing Machine is TM_2) s_2, it need only be proven that the vertices outputted by TM_1 is a subset of vertices outputted by TM_2 and the shortest distances in the corresponding vertices are the same. Based on the non-decreasing monotonicity of the vertex order on their shortest distances, s_1 is always compatible with s_2. Since the vertices in s_1 are vertices in S and D plus all the boundary vertices, it is always a subset of the vertices in s_2 which are the vertices in the digital map. Also it is obvious that the subsequence before the first boundary vertex in s_1 is equivalent to the same sized prefix of s_2 because both of which are the results of applying Dijkstra's algorithm on fragment S. Suppose that v is the first vertex in s_1 such that $s_1(-, v)$ is compatible with $s_2(-, v)$, and $SD_{s_1}(v) > SD_{s_2}(v)$. ($SD_{s_1}(v)$ cannot be less than $SD_{s_2}(v)$ because $SD_{s_2}(v)$ is always the optimal solution to the shortest path problem). Suppose u is the preceding vertex to v in s_1, by assumption $SD_{s_1}(u) = SD_{s_2}(u)$, we know that u is the parent vertex of v in the shortest path in TM_2. From Dijkstra's algorithm, we know that $SD(v) = SD(u) + SD(u, v)$, where $SD(u, v)$ is the shortest distance between u and v in their common fragment, say F_1. Suppose that all vertices in the subsequence $s_2(u, v)$ are in fragment F_1, which means the actual optimal shortest path from u to v in the digital map is within fragment F_1. Therefore, this should have been $SD_{s_2}(v) - SD_{s_2}(u) = SD_{s_1}(v) - SD_{s_1}(u)$, which is contradictory to $SD_{s_1}(v) > SD_{s_2}(v)$. The other case is that there are vertices in other fragment(s) in the subsequence $s_2(u, v)$, which means that the actual optimal shortest path from u to v is totally within fragment F_1. If there is no boundary vertex in the subsequence $s_2(u, v)$, which implies u and v are in the same boundary set, say $[F_1, F_2]$ (otherwise the shortest path must pass though

at least one other boundary vertex to reach v from u), and all vertices in $s_2(u, v)$ are in fragment F_2. Therefore, when u is closed, v is relaxed for both fragments F_1 and F_2. Based on the Dijkstra's algorithm, we have $SD_{s_1}(v) = \min(SD_{s_1}(u) + SD_{F_1}(u, v), SD_{s_1}(u) + SD_{F_2}(u, v)) = SD_{s_2}(v)$, which leads to a contradiction. If on the other hand, there are boundary vertices, say $w_1, \ldots, w_k$, in $s_2(u, v)$, they should be in $s_1(-, u)$ or $s_1(v, -)$, otherwise it is contradictory to that fact that u is the preceding boundary vertex of v in s_1. We assume that $SD_{s_1}(w_i) \neq SD_{s_1}(u)$, otherwise these two sequences are equivalent and thus would be compatible already. If w_i is in $s_1(-, u)$, we have $SD_{s_1}(w_i) = SD_{s_2}(w_i) < SD_{s_1}(u) = SD_{s_2}(u)$, which is contradictory to the assumption $SD_{s_2}(w_i) > SD_{s_2}(u)$. If w_i is in $s_1(v, -)$, we have $SD_{s_1}(w_i) > SD_{s_2}(v) \geq SD_{s_1}(u) \geq SD_{s_2}(u) \geq SD_{s_2}(w_i)$, which means in TM_2, the boundary vertex w_i is relaxed when u is relaxed which is contradictory to the algorithm. Therefore, all possibilities lead to a contradiction, hence the conclusion is that there is no boundary vertex in s_2 such that its shortest distance value is different from that in s_1. Therefore, TM_1 and TM_2 are always generating the same results on the same input graph.

□

3.6.5 Complexity Analysis

The complexity of the disk-based shortest path algorithm is two-fold: the CPU complexity, and the I/O complexity. Much of the CPU complexity comes from the update of the main memory data structure and relaxation. The I/O complexity is more complicated since different buffer management schemes result in different I/O

performance results. Therefore, for I/O complexity, the worst case is calculated, i.e. assume there is no buffer. Every time the algorithm accesses a disk-based data structure, it results in an I/O operation. Later in Chapter 5, the empirical results of the effects of the buffer management scheme will be shown.

In Algorithm 6, there are three types of heaps: a binary heap for vertices in S and D, a U-Heap containing the delegate vertices of all boundary sets, and a Fibonacci heap for vertices in each boundary set. The first two types of heaps are stored in main memory; the Fibonacci heaps are stored on disk but ready to load into main memory when it is required. Therefore, the CPU complexity consists of the manipulations of these three types of heaps, plus the relaxation process for the vertices in S and D, as well as in the $MainThrust$ process. For the manipulation of binary heaps, the CPU complexity is the same as Dijkstra's shortest path algorithm on S and D. Therefore, it is $O(n \log n + m)$, where n is the maximum number of vertices in a fragment, m is the sum of number of edges in S and D. For the manipulation of U-Heap, the worst case is that you have to update the U-Heap every time when you have done a relaxation on a boundary set. A boundary set is relaxed every time a boundary vertex in the two adjacent fragments is ready to close. Assume that the number of boundary vertices in a fragment is b, and there are s boundary sets. Then each boundary set is relaxed $2b$ times, so the total times of relaxation is $2b * s$. For each relaxation, the U-Heap updates the value of a leaf node. Then the ancestors of the leaf nodes should also be updated if necessary. The worst case is that all the ancestors are updated, so the update takes $(\log s)$ times. Therefore, the running time complexity of U-Heap is $O(b * s * \log s)$.

For the manipulation of Fibonacci heap, the worst case is that the $decreaseKey$ operation of a node must be done every time it is relaxed. Since the maximum times a boundary vertex begin relaxed is $2b$, and there are $b * s$ boundary vertices, the maximum times of $decreaseKey$ operations in a Fibonacci heap is $2*b^2*s$. Since the amortized running time of $decreaseKey$ operation in the Fibonacci heap is $O(1)$, the total complexity of $decreaseKey$ operation is $O(b^2 * s)$. Another operation in the Fibonacci heap is the $extractMin$ which runs in $O(\lg b)$ amortized time. The total number of $extractMin$ operations is the number of boundary vertices $b * s$. Therefore, the complexity of $extractMin$ operations in the Fibonacci heap is $O(s * b * \lg b)$. Since the number of summations and comparisons in the relaxation is the worst case of number of $decreaseKey$ operations in the Fibonacci heap, its complexity is in a lower order of the complexity of $decreaseKey$ operations. Therefore, summing up the complexities of the three types of heaps, we get the total CPU complexity as follows:

$$O(n \log n + m + bs \log s + sb \lg b + sb^2) = O(n \log n + m + sb^2 + bs \log s)$$

For the I/O complexity, it is assumed that every time we access a disk-based data structure, there will be a B_i-bytes I/O operation, where B_i is different for different disk-based data structures. Therefore, the I/O complexity can be simplified by two measurements: the number of I/O's and the number of bytes being read in and written out. The disk-based data structure in the $DiskSP$ algorithm is used for containing distance vectors for boundary sets, the fragment database and the distance matrix database. The distance vector is accessed whenever its

corresponding boundary set is relaxed. By the analysis of CPU complexity, it is known that there are totally $2b*s$ relaxation of boundary sets. Since the relaxation of boundary vertices in a boundary set can be batched, it is possible to have only one I/O for each relaxation of a boundary set. The I/O of a distance matrix is also during relaxation. Each distance matrix is accessed once whenever its corresponding fragment has a boundary set to relax in the worst case. Therefore, the number of distance matrix access is the same as the number of relaxation of boundary sets, i.e. $2b*s$. The number of I/O of fragment database depends on how many fragments the resulting shortest path traverses, since the fragment database does an I/O in the *FillSP* algorithm, i.e. to complete the simplified shortest path by applying Dijkstra's shortest path to the intermediate fragments. The worst case is that every fragment is traversed once. Therefore, the number of I/O is the number of fragments, i.e. in $O(s)$. Therefore, the total number of I/O is in

$$O(s + b * s + b * s) = O(bs)$$

the total bytes of I/O is in

$$O(sB_1 + bsB_2 + bsB_3)$$

where B_1, B_2, B_3 are the numbers of bytes being read in or written out by a fragment database, distance vector and distance matrix data structures respectively.

Chapter 4

Implementation

The system architecture, testing data source, and the implementation details on the data structures and the algorithms are discussed in this chapter.

4.1 System Architecture

Java was chosen as the implementation language because Java is a fully object-oriented and a fast prototyping programming language. Java is also a network-centric language. The program can be easily migrated to multi-tier application architecture shown in Figure 4.1

In this architecture, the server side is divided into three tiers: Web Server and GUI server, Route Query Engine, and Spatial Database Server. The Web server and GUI Server tier is responsible only for receiving the users' requests and displaying the shortest paths results; the Route Query Engine tier does the actual route planning job; the Spatial Database tier provides the data required by the

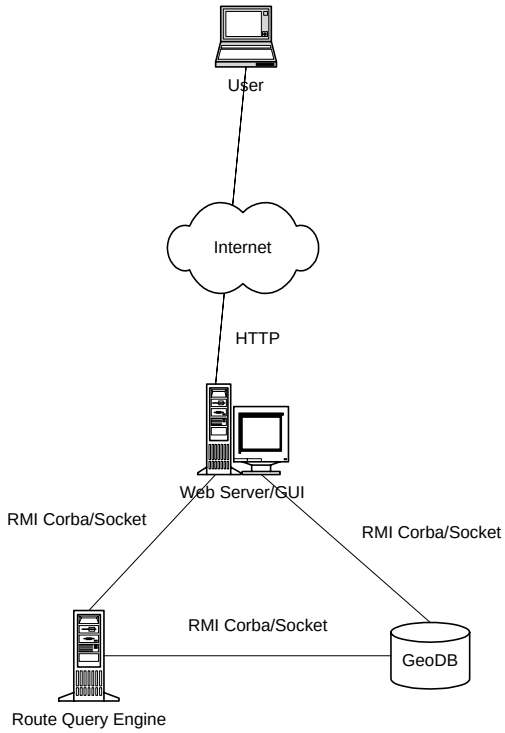

Figure 4.1: Route Query Application Architecture

Route Query Engine tier and the GUI server, and is also responsible for updating the volatile data (such as edge weights and path views). With this architecture, a route-planning query goes like this:

- The end users submit route-planning queries through a web browser or other client programs.
- The GUI server gets queries from the Web server and calls the services provided by the Route Query Engine.
- The Route Query Engine receives the queries, parses them, calls the services provided by the Spatial Database Server to get the appropriate geometric

data, and then launches the appropriate route-finding algorithms on the data to find the path. At last, the Route Query Engine returns the final results to the GUI server.

- The GUI server then calls the services provided by the Spatial Database Server to get the geometric data for the resulting paths, which are used to render the map to the end user through the Web server.

The tasks of each server will be defined in Section 4.2. Determining how to call the services provided by them is also necessary. Using Java, there are three approaches on the candidate list. The first is to use Socket programming. In this approach, a server is a daemon program listening to a particular port. The second choice is to use Java Remove Method Invocation (RMI), which is an object-oriented and platform independent approach. The third is to use Common Object Request Broker Architecture (CORBA), which is an independent language as well as having the advantages of RMI. The disadvantage of CORBA is that it is considerably slower and more complicated to implement. The socket approach is the fastest of the three, but it takes extra efforts to define the communication protocols and to manipulate the data. The RMI approach may best fit this kind of application, since all our programs are written in Java. Therefore, it is not necessary to sacrifice efficiency for the language independency that CORBA pursues. It also provides remote objects and method calls which is a very nice feature and great advantage over socket programming. In the following discussion, we define the services provided by servers as remote objects and methods. By defining the objects and methods in them, we can know precisely what services could be acquired from that server. The services

and API's will be introduced in section 4.2.

The advantages of this architecture is that

1. It is highly portable and platform independent.

2. By distributing the task to different servers, better performance and scalability is attained.

3. This multi-tier system can be upgraded more easily than client-server architecture. Since tasks are distributed to different servers according to their functions, the corresponding servers can be upgraded without affecting other servers. For example, for dynamics query cases, the weights of edges can change at any time. Therefore, it is better to store the edge weights in a different database rather than the relatively stable digital map databases. Maintaining the edge weights only affects the spatial database server, not the route query engine or web server.

4. Hardware resources are not as demanding as client-server architectures thus the costs could be lowered. Since both the route query engine and the spatial databases are resource consuming (route query engine requires a powerful CPU and spatial databases prefer faster I/O processing time), dividing these two sub-tasks into two servers is preferable. One has a faster CPU but might have a slower I/O; the other has a faster I/O ability but might not have as fast a CPU.

However, this architecture also has disadvantages:

1. There is communication overhead between servers in different tiers. Part of the overhead comes from the data transfer via the network; the other part comes from the overhead over the software (RMI or CORBA). Nevertheless, with the emerging high speed and broadband networks, network communications are even faster than hard disk I/O. With operating systems and other low level software accommodating the hardware changes, the networks not longer need to be considered as bottlenecks. As for overheads caused by RMI and CORBA, some techniques have been developed to improve the performance, such as caching techniques.

2. Multi-tier applications are more complex than client-server architecture in terms of implementation. Since the application is distributed over the network, more work should be performed on communication, synchronization and management than the client-server applications. Nevertheless, with the new technologies such as Enterprise Java Beans (EJB) and XML, programming on distributed environments and managing distributed objects can be much easier. Java is the most leading-edge language that provides such services at the date this thesis is written.

4.2 Data Sources

For GIS systems, the data could come from various data sources. In the United States, the geographical information is maintained in different ways. For example, the Bureau of The Census maintains a database called Census TIGER (Topologi-

cally Integrated Geographic Encoding and Referencing) [43]. The TIGER/Line files are extracts from the TIGER database of selected geographic and cartographic information. It contains the line segments that represent physical features, and legal and statistical boundaries. The files consists of 17 record types, including the basic data record, the shape coordinate points (feature shape records), and geographic area codes that can be used with appropriate software to prepare maps. From the 17 record types, the data can be divided into three major types of features:

- Line features including roads, railroads, hydrography, miscellaneous transportation features and selected power lines and pipelines, and boundaries.
- Landmark features including point landmarks such as schools and churches, area landmarks such as parks and cemeteries, and key geographic locations (KGL) such as apartment buildings and factories.
- Polygon features including geographic entity code for areas used to tabulate the 1990 census statistical data and current geographic areas, locations of area landmarks, and locations of KGL.

In terms of network queries in the Spatial Database systems, we do not have to worry about the landmark features and polygon features, so we can extract the line features from the TIGER/Line files only. In order to be able to deal with multiple types of data sources, it is better to extract the common properties of line features in various data sources and construct an abstract layer which is called Abstract Line Feature Layer (ALFL), as shown in the Figure 4.2 below.

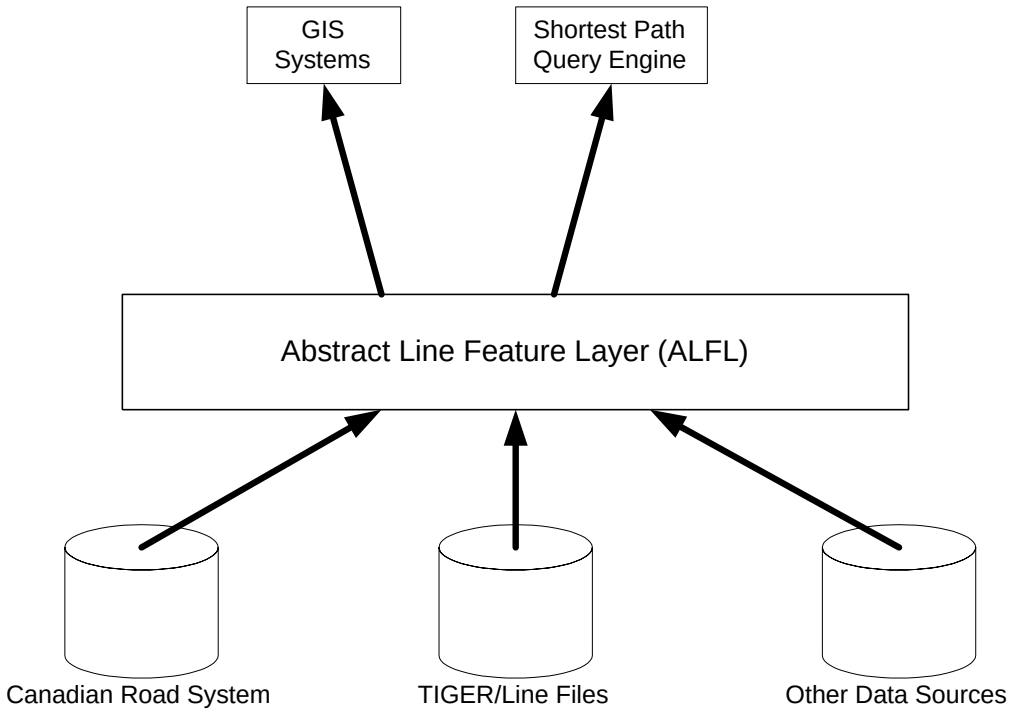

Figure 4.2: Geographical Line Features Abstraction

It is only necessary to program on the ALFL in our shortest path query engine without worrying about the specific data source we are working on. The ALFL actual hides the differences of the data sources, and acts as an interface to the upper layer software.

In this implementation, the ALFL is a set of tables in relational database systems. By defining the columns and constraints on the columns on different tables, the ALFL interface can be defined in a precise way. The tables and their relationships are shown in Figure 4.3.

In the Abstract Line Feature Layer, the StreetBlocks table plays a central role by storing all street blocks (line features) in one table. Each street block is uniquely

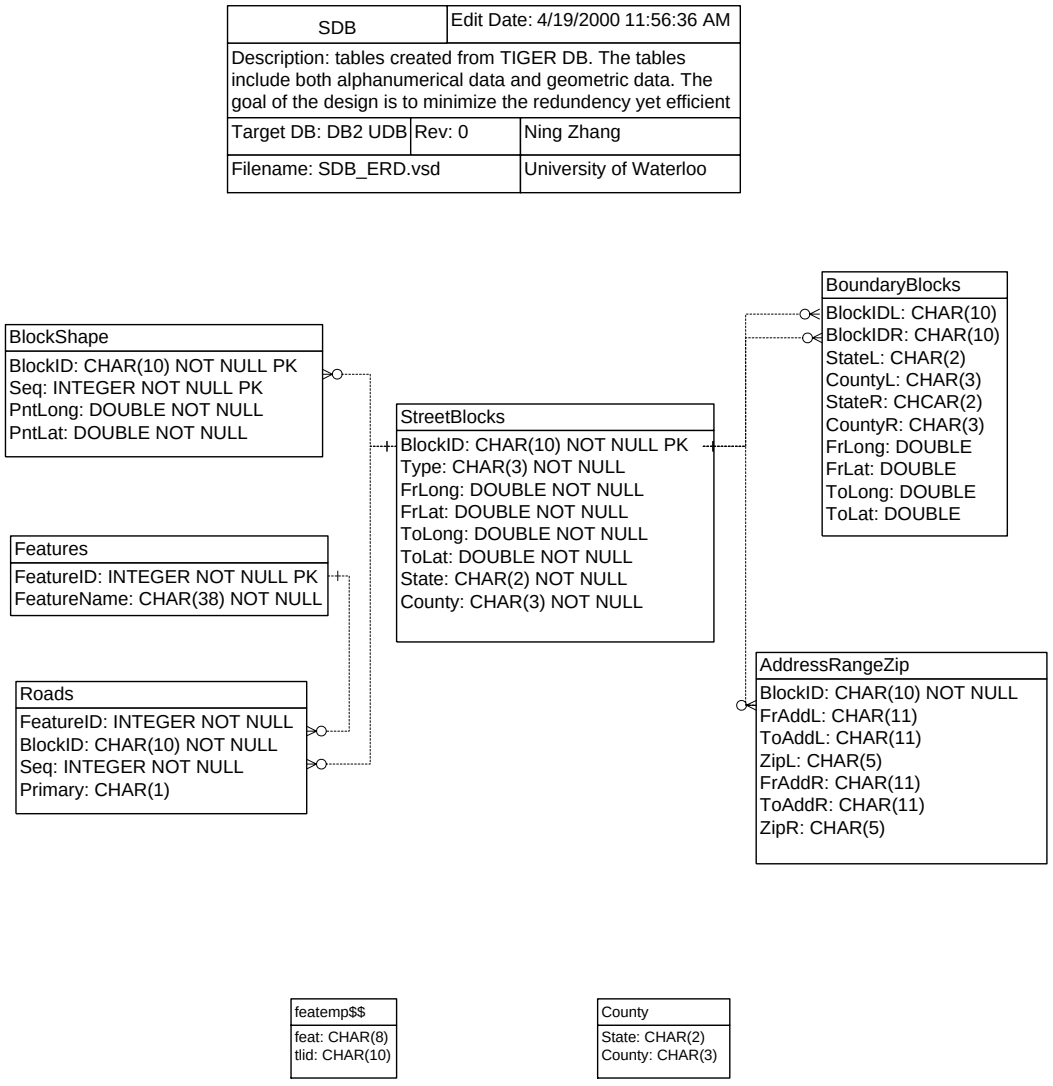

Figure 4.3: Abstract Line Features Layer Tables

identified by a block ID which is a 10-character length string. If the roads are classified, the type of the street block (interstate highway, state highway or local road) is captured in the Type field in the StreetBlocks table. The longitude and latitude of the "from" node or "to" node (FrLong and FrLat, or ToLong and ToLat

fields) in the StreetBlocks table uniquely determines an intersection in the road system. The State and County fields hold which state and county this street block is located in. If a street block lies exactly on the border of two counties, the State and County fields are set to a special value ("00" and "000" in this implementation). In this case, the StreetBlocks table must be joined with the BoundaryBlocks table to find the state and county information on both sides. The fields BlockIDL (block ID on the left hand side starting from "from" node to "to" node) and BlockIDR (block ID on the right hand side) are the proper foreign keys to the join operation.

If the shape of a street block is not a straight line, it is simulated by a sequence of ordered straight lines with one's head being another's tail. The coordinates of the intermediate points (shape points) are given in the BlockShape table. The BlockID field in the BlockShape table acts as a foreign key to the BlockID in the StreetBlocks table. The seq field in the BlockShape table indicates the order of the shape point in the street block. The less the seq, the closer it is to the "from" node.

The Roads and Features tables together capture the road features in the transportation system. A road feature is defined to be a connected sequence of street blocks that have the same feature name. Since feature name itself cannot uniquely determine a road (for example both Waterloo and Toronto have a King Street), a unique feature ID field was introduced to identify the road features. The seq field indicates the order of a certain street block in a certain road feature, while which street block in the road feature is the starting block, is undefined. Some roads may have more than one name. In this case, the same road may belong to more than one feature, but one feature name must be its primary name. This information (a

single character 'Y' or 'N') is kept in the primary field in the Roads table indicating whether the feature name is its primary name or not.

Address ranges and zip code information is stored in table AddressRangeZip that can be joined with StreetBlock table by the foreign key BlockID. For one street block, the addresses on either side may not constitute only one range. It is possible (at least in theory) that the address ranges of two street blocks interleave. In this case, we have to break down the address range into finer ranges. For each address range, there is a unique zip code associated with it. This is the constraint on the input of this table.

The County table is an auxiliary table that keeps track of which counties have been processed and stored in the tables. It is useful when it is necessary to incrementally insert data from TIGER table to the ALFL tables. The featemp$$ table is another temporary table for optimizing the query performance when populating the Features and Roads table from the StreetBlocks table.

In this implementation, another program reads the data from the ALFL tables and constructs a geometric database consisting of geometric objects, and then a Hilbert R-Tree can be built on the geometric database, both of which, can be applied to our graph partitioning algorithm. The process procedure is shown in Figure 4.4 as follows.

In respect to the figure above, the program used to convert the TIGER/Line files to ALFL tables is called MakeSDB.java. It is necessary to provide the TIGER/Line file names and the database name for storing the ALFL. Converting ALFL to the Geometric Database is done by the Sdb2Gdb.java. For this program, it is necessary

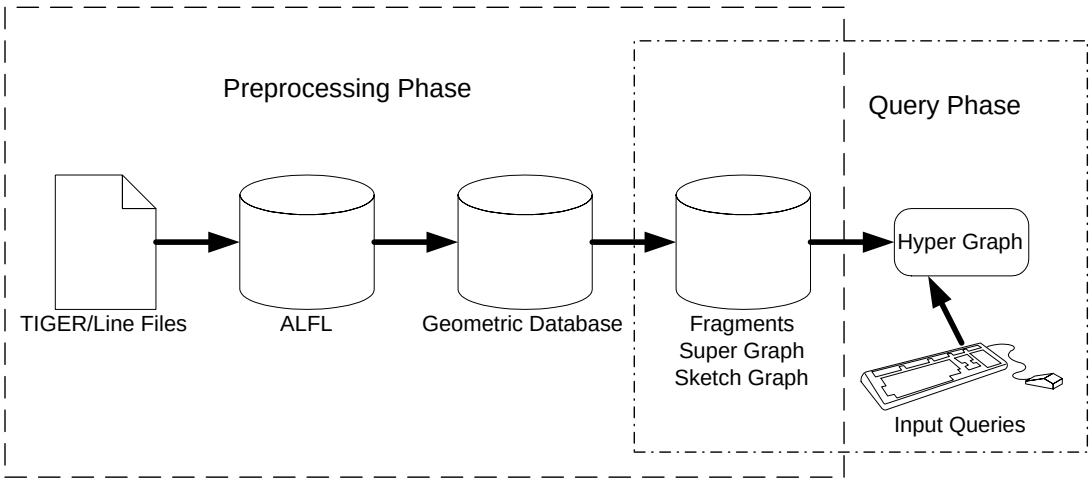

Figure 4.4: Data Flow

only to provide the database name. From the geometric database to the fragments database, it is necessary to invoke the graph partitioning algorithm, which is implemented in the HRTreePartition.java. The program takes four arguments, the geometric database name, as well as the Hilbert R-Tree, the minimum and maximum number of vertices in the resulting fragment, and the name of the resulting fragment database. The super graph database and the sketch graph are generated by the program MakeSPDB.java by taking the fragment database as the input.

After these processes, the preprocessing phase is over and the input queries are ready to be accepted. The input query is composed of a pair of source and destination vertices. With the input vertices and sketch graph, pruning on the sketch graph can be done, and then pruned sketch graph can be generated. At last the Dijkstra's shortest path algorithm is applied on the pruned sketch graph and the shortest path connecting source and destination can be found. The implementation details of the pruning algorithm are described in section 4.3.2.

4.3 Graph Representation and Class Hierarchical Structures

Bearing the system architecture in mind, we should design the program such that as little data as possible is transferred from one system to another. The classes and their relationships are shown in UML in Figure 4.5.

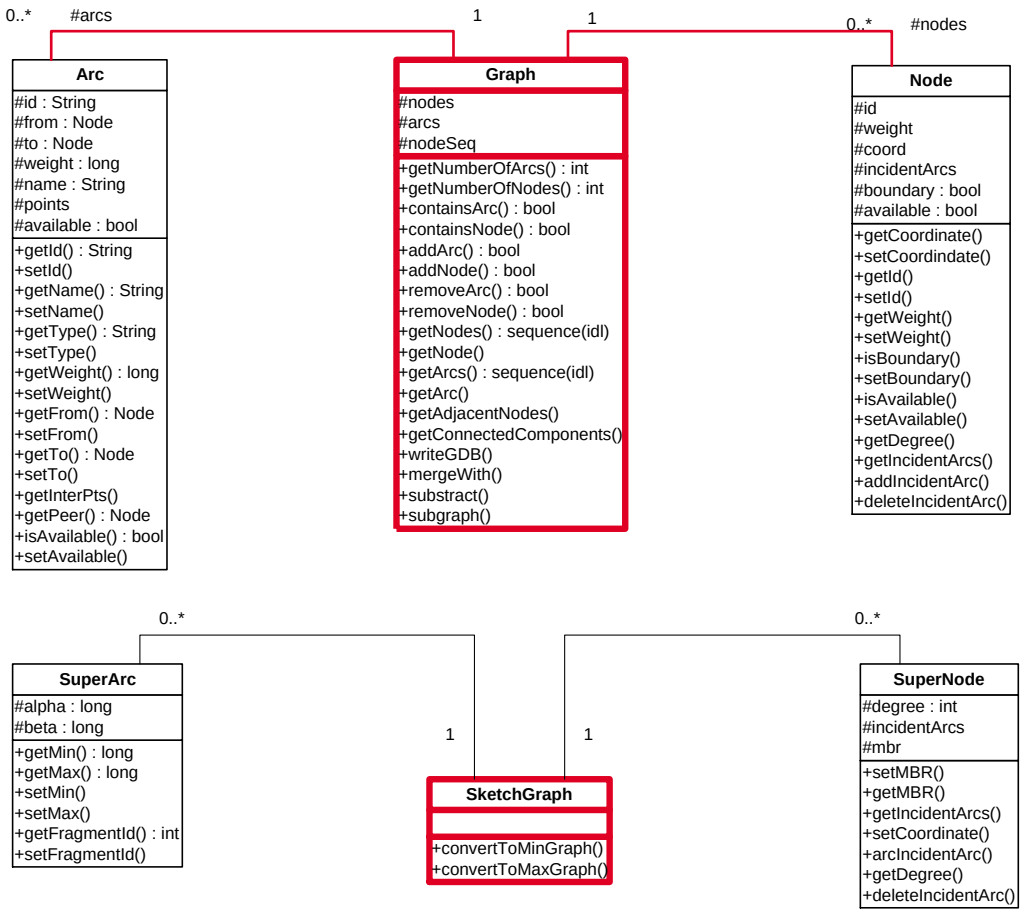

Figure 4.5: Graph and SketchGraph UML

In this implementation, each fragment can be viewed as a graph object, which

consists of a set of arcs and a set of nodes. Both of them are stored in hash tables. For the node hash table, the keys are the ID's of the nodes, which are their coordinates. For the arc hash table, the keys are also their ID's, which are unique 10-character ID's. Since currently we use TIGER/Line files as our input, the ID of an arc is already given and is guaranteed to be unique for the data in the United States. If other data sources are used and they do not provide unique ID's for arcs, they must be assigned unique ID's.

When the boundary vertices are pushed up to the higher level and form a super graph, the super graph itself can also be represented by a graph object. However, when the sketch graph is extracted from the super graph, it cannot be represented by a graph object any longer. SketchGraph is a separate class containing SuperArcs and SuperNodes instead of Nodes and Arcs. A SuperNode represents a boundary set, while a SuperArc represents the all pair shortest paths from one boundary set to another boundary set in some fragment. In SuperNode, we record the MBR of the boundary set. In SuperArc, we record the minimum and maximum shortest distances (alpha and beta respectively) from some vertex in one boundary set to some vertex in another boundary set. Since we have two special properties a and b associated with each super edge in the SketchGraph, we define two methods in the SketchGraph: convertToMinGraph() and convertToMaxGraph(). The first method converts the SketchGraph to a general Graph object with the edge weight being the a value. The second method does a similar processing except that the edge weight is the b value. These two methods are useful when vertical pruning to the sketch graph is done to the sketch graph.

4.4 Implementation Details

In this section, the implementation details about the graph partitioning algorithm based on BFS, the vertical pruning algorithm, the disk-based shortest path algorithm and the virtual data structures are introduced. Particular source codes are included with comments.

4.4.1 Building the Shortest Path Query Engine Step by Step

Now a detailed description is given of how to build the database for a shortest path query from raw TIGER/Line files. There are two phase: pre-processing phase and querying phase, and seven steps totally, in the two phases:

1. Pre-processing phase: build up the databases for shortest path query. There are five steps in this phase:

   Building up TIGER database from TIGER/Line files. Sample command: java Library.Route.MakeTigerDB tgr09001 NYC
   This command insert the data in the raw TIGER/Line file "tgf09001" into a TIGER database contained in the data source "NYC". If you want to insert multiple TIGER/Line files into the same TIGER database, just invoke the command multiple times with different arguments.

   Building up ALFL database from TIGER database. Sample command:

```
java Library.Route.MakeSDB NYC
```

This command convert the TIGER database in NYC to ALFL database in the same data source. Note that a data source can contain different databases (set of tables) at the same time.

Building up a Geometric Object Database (GDB) and the Hilbert R-tree on them. Sample command:

```
java Tools.Sdb2Gdb NYC
```

This command reads the line features from the ALFL database, converts them into geometric objects, stores the geometric objects in a geometric database, and builds the Hilbert R-tree index on the GDB.

Partitioning the GDB based on the Hilbert R-tree into fragments and stored in a fragment database. Sample command:

```
java Library.Route.HRTreePartition NYC 15000 20000 NYC\_frag.db
```

This command partitions the geometric objects contained in the GDB NYC into fragments, which are stored in the file "NYC_frag.db". the two figures 15000 and 20000 specify the range of (minimum and maximum respectively) number of vertices of the fragments when partitioning.

Calculating the k-pair shortest paths for all fragments in the database and store the shortest distances in a Distance Matrix Database. Meanwhile, the sketch graph for the fragments are also generated and materialized to hard

disk. Sample command:

```
java Library.Route.MakeSPDB NYC\_frag.db
```

This command computes the k-pair shortest paths for every fragment in the database "NYC_frag.db". The distance matrix database is stored in a default file "DistMatrix.db". The sketch graph is serialized into a default file "sketch.ser". And the boundary sets of the fragments are stored in another file "BoundarySets.db" for ease of loading boundary sets without loading a whole fragment.

2. Querying phase: accept user's shortest path query, and return the result. There are two steps in this phase:

   Pruning the sketch graph by the given pair of vertices, and generate a pruned sketch graph. Sample command:

```
java Library.Route.PrunedSP sketch.ser NYC\_frag.db  (-73.249459,41.367495) (
```

This command takes the two vertices represented by their coordinates in the form of inside parentheses as source and destination, and tries to prune the sketch graph which is serialized in "sketch.ser". The fragment database "NYC_frag.db" is used to find the fragments in which the source and destination vertices are. If the information is given by other processes, the fragment database argument is not necessary.

Finding the shortest path by the pruned sketch graph as well as the Distance Matrix Database and the fragment database using the disk-based shortest path algorithm. Sample command:

```
java Library.Route.PrunedSP pruned.ser ct\_frag.db   (-73.249459,41.367495) (
```

This command is taking the same source and destination vertices as those in the above pruning step and the pruned sketch graph, as well as the fragment database as input, finds the shortest path between these two vertices.

In the above seven steps, not all of them are executed once whenever there is a new shortest path query submitted. The five steps in the pre-processing phase should be executed only once for static shortest path queries. When new source and destination pairs come, only the last two steps in the querying phase are executed. Therefore, our algorithm should optimize these two steps as far as possible.

4.4.2 Graph Partitioning Algorithm

The purpose of the graph partitioning algorithm is to divide the digital map into fragments such that they are small enough to be read into main memory. The fragments are stored into fragment databases by a virtual data structure. Since super graphs must be constructed based on these fragments, the boundary vertices should also be found out and attached to the fragments. Based on this requirement, at least two classes are needed: Fragment class and BoundarySet class. A Fragment class extends from a Graph class, but with some extra properties: a unique fragment

ID (integer), the MBR of the fragment, and a set of BoundarySet of this fragment. BoundarySet class is the set of boundary vertices shared by and only by certain fragments. The ID of a boundary set is the ordered sequence of the fragment ID's. For example, fragments 1, 3 and 10 are adjacent fragments, and their vertices are v_1, v_2, v_3, v_4, v_3, v_4, v_5, v_6, and v_3, v_5, v_7, v_8 respectively. There are three boundary sets $< 1, 3 >$, $< 3, 10 >$, and $< 1, 3, 10 >$. Boundary set $< 1, 3 >$ contains vertices $v4$, which are common vertices shared by fragments 1 and 3. Likewise, $< 3, 10 >$ and $< 1, 3, 10 >$ contains $v5$ and $v3$ respectively. Note that v_3 is not in the boundary set $< 1, 3 >$ nor $< 3, 10 >$ since it is a common vertex in fragment 1, 3 and 10, so fragment 1 and 3, or 3 and 10, are not the only fragments contains v_3. In this example, boundary sets $< 1, 3 >$ and $< 1, 3, 10 >$ are associated with fragment 1; $< 1, 3 >$, $< 3, 10 >$ and $< 1, 3, 10 >$ are associated with fragment 3; and $< 3, 10 >$ and $< 1, 3, 10 >$ are associated with fragment 10.

Since the digital map is too large, part of the map must be read first into memory and then explored using BFS. During the graph traversal, part of the graph in memory can be removed and other parts could be read into memory and merged to the existing one. According to the algorithm given in Section 3.2.1, the digital map should be partitioned into small grids first and then the grids should be merged as the algorithm goes. With Hilbert R-Tree, partitioning the digital map into grids is very easy, just by calling the method query(mbr, objs) in the HRTree class. The first parameter mbr is the MBR which will retrieve the geometric objects. The second parameter objs is an empty vector object that when the method returns, contains the file pointers of the geometric objects in the geometric database file.

Then the geometric objects can be read from the database according to the file pointers, and assembled into a graph object.

In order to keep track of which grids have been read into memory and merged, we define a Grid inner class as follows:

```
class Grid
{
    sdbRectangle mbr;
    boolean untouched; // is it already explored?
    Grid(sdbRectangle rect)
    {
        mbr = rect;
        untouched = true;
    }
}
```

It contains two fields: mbr and untouched. The mbr field records the rectangle of the grid. The untouched field keeps the status of the grid. The grid formatted digital map can be illustrated as in Fig. 3.2.

Suppose one vertex was selected in grid 0 as our starting point for traversing. Since the next vertex could possibly be in grids 1 to 8, grids 1 to 8 must be read and merged into a graph object before traversing. Merging the eight neighbour grids (north, northeast, east, southwest, south, southwest, west, northwest) of a center grid is called preparing-graph. This process guarantees that the BFS behaves the same way on the partial digital map as on the digital map itself. Each grid of the digital graph has a Grid object in main memory. When a grid is read and merged, the untouched field in the corresponding Grid object is set to be false. When a new vertex is explored, its grid is examined to see if all of its eight neighbours have been "touched". When a fragment is constructed, it has to be removed from the graph object in memory, otherwise the graph object will grow too large.

4.4.3 Vertical Pruning Algorithms

For the vertical pruning algorithm, the inputs are the sketch graph and a pair of source and destination vertices. The output is the pruned sketch graph. Given the pruned sketch graph together with the fragment database and the distance matrix database, the shortest path between the source and destination pair can be easily found using the disk-based shortest path algorithm.

The vertical pruning algorithms are always used together, so they are bounded into one method PrunedHyper() in the GraphAlgorithms class in this implementation. They work like a pipeline as shown in Figure 4.6.

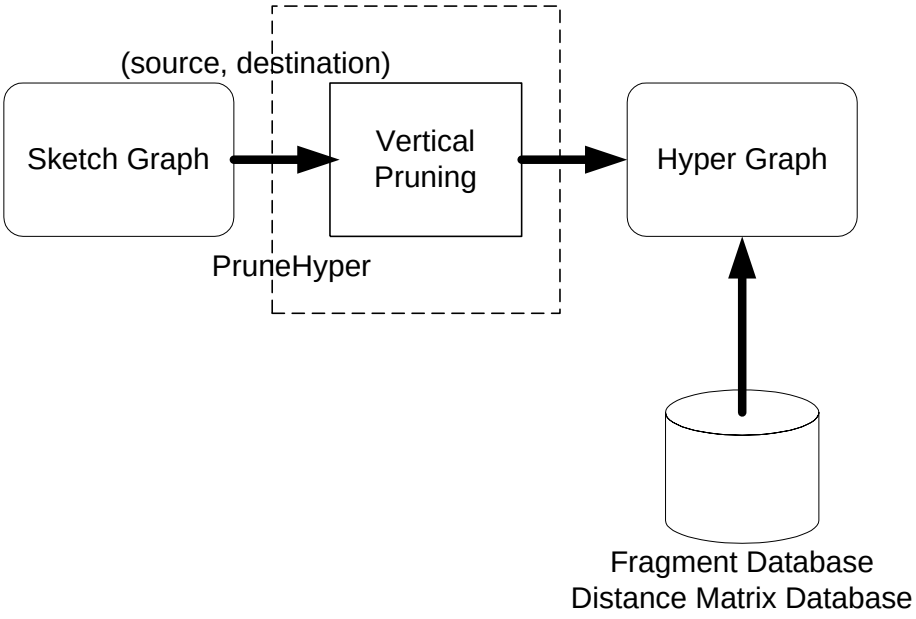

Figure 4.6: Pruning Sketch Graph

4.4.4 Virtual Data Structures

Figure 4.7 depicts the internal of virtual data structures.

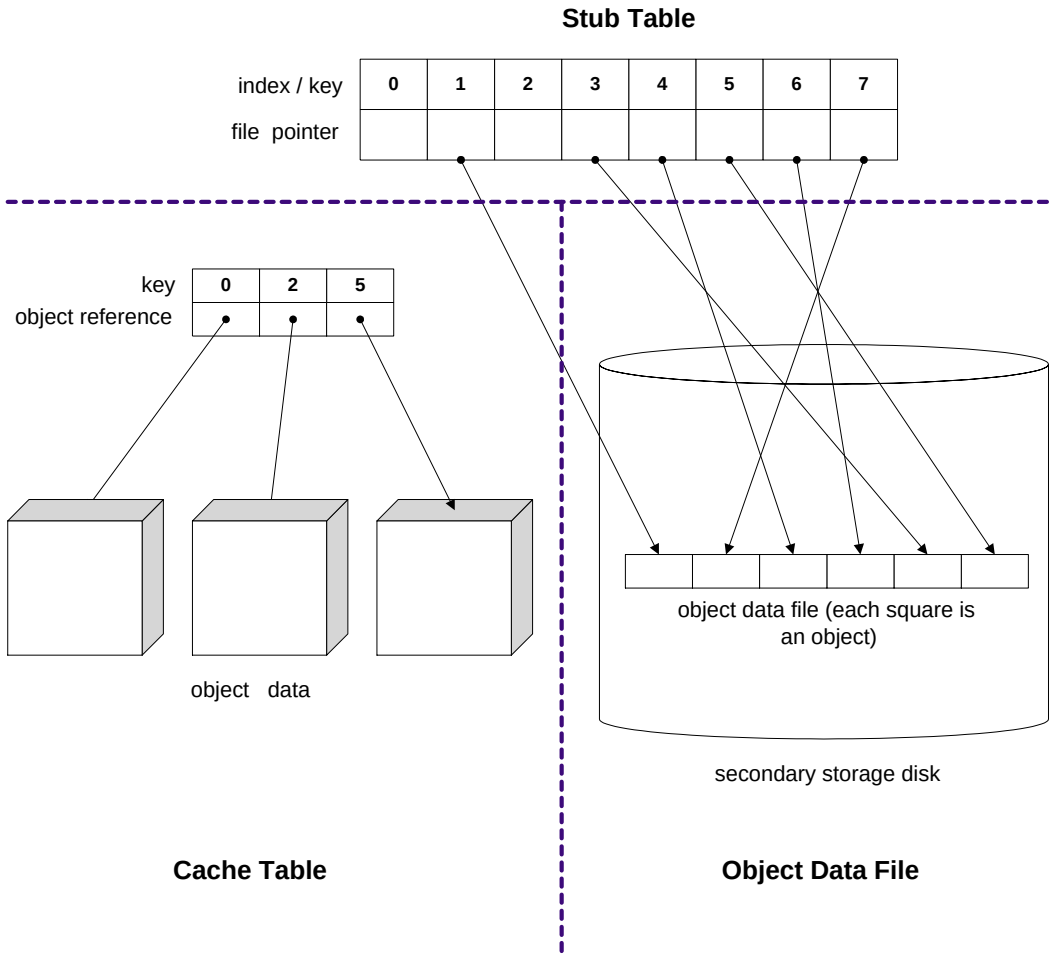

Figure 4.7: Virtual Data Structure Diagram

Inside the virtual vector and virtual hash table, there are two tables maintained by the data structure in main memory; a stub table recording all data entries in the virtual vector or virtual hash table, and a cache table for boosting performance

by buffering a small number of objects in main memory.

The entry in the stub table is simply a 3-tuple – (InCache, Key, FilePointer), where InCache is a Boolean variable indicating whether this data entry is in the cache table or not. In Figure 4.7 the shaded entries 0, 2, and 5 are in cache while others are not. FilePointer indicates the location in the file where the object data should be read or written out. Some entries in the stub table may not have file pointers such as entries 0 and 2 in Figure 4.7. This can only occur to entries in cache because of the "lazy synchronization" policy used. With lazy synchronization, a newly added object is not written to the object data file immediately. Rather it will be put to the cache first. When the object is going to be swapped out or the virtual data structure is going to be closed, it is written out to the object data file. Also because it is not known whether the object has been modified by the programmer or not, when it is written out, we have to check the size of the object. If its size is the same as it is recorded in the object data file, the object data is written to the same location as where it is read, otherwise the object should be appended to the end of the file and the previous copy, if any, should be marked as removed. In the virtual data structure, each entry has an associated key boosting search operations. In the virtual vector, the stub table is a vector, so the key is the index of the stub vector; while in the virtual hash table, the stub table is a hash table, and we simply use the key of the hash table as the key of virtual hash table.

The entry in the cache table is a 2-tuple - (Key, Object). The first field has the same meaning as the "key" field in the stub table. Therefore, the key set in the cache table is a subset of keys in the stub table with the InCache flag being true.

The other field is an object reference to the actual object data. We use the Least Recently Used (LRU) algorithm to determine which entry should be swapped out from the cache buffer.

In the Java implementation, the cache buffer does not have a fixed size, although a maximum buffer size can be specified. After version 1.2, JDK provides a useful class called SoftReference. You can instantiate an object and point it with a SoftReference. When the Java Virtual Machine runs out of memory, the objects pointed by only soft references are guaranteed to be reclaimed before an OutOfMemory exception is thrown. This feature is very useful in memory management. In the virtual data structure, the objects in cache can be assigned to SoftReferences. Therefore, you do not have to worry about the buffer size being too large for certain database, since the JVM will do the swapping automatically if necessary. However, another difficulty arises if the JVM determines which object should be eliminated from the cache: it is necessary to synchronize the object to the object data file before it is reclaimed. Fortunately, Java offers a finalize() method allowing some destructive work to be done before the object is reclaimed. Therefore, the finalize() method must be overwritten in the cache entry class to ensure the consistency of the object data file with the cache. The snippet of finalize() method for CacheEntry is show as follows:

```
protected void finalize()
{
        // we don't deal with the the CacheEntry that
// is already removed from the cache.
        if ( ! ((StubEntry)rows.get(key.intValue())).inCache )
            return;
        else if (cache.get(this.key).hashCode() != this.hashCode())
            return;
```

```
        // manipulate StubEntry table.
        try {
            StubEntry se = syncEntry(this); // write out data to object
                                            // data file if necessary
            se.inCache = false;
        } catch (IOException e) { e.printStackTrace(); }

        // manipulate LRU doubly linked list
        if ( this == firstCacheEntry )
            firstCacheEntry = next;
        if ( this == lastCacheEntry )
            lastCacheEntry = prev;
        if ( prev != null )
            prev.next = next;
        if ( next != null )
            next.prev = prev;

        iCacheSize--;
    }
```

In this implementation, an object in the cache is first serialized to a byte stream, and then the length of the byte stream is written to the file followed by the byte stream itself. For the virtual vector, when an existing object data file is opened, the stub table can be constructed on the fly by reading through all objects in the file. Therefore, no further information needs to be stored in the object data file. However, for the virtual hash table, the keys of the stub table cannot be obtained by reading through the objects themselves. The only way is to store the keys also in the object data file. In this implementation, the stub table itself is also serialized and appended to the end of the object data file, followed by the length of its byte stream. In this way, when a virtual hash table is opened, the last four bytes in the object data file, which is the length of the stub table, are read first. Then read in the stub table from the end of the file. The file format is show as Figure 4.8

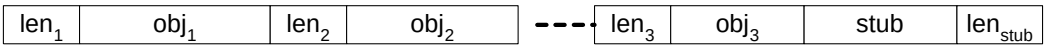

Figure 4.8: Object data file for Virtual Data Structure

4.4.5 Disk-based Shortest Path Algorithm

In the DiskSP algorithm, there are five data structures used: binary heap, U-Heap, Fiboancci heap, distance vector, and distance matrix, each of which is implemented by a class in Java.

Details in Binary Heap

The binary heap is implemented by an array as suggested in [1], because accessing elements in an array is much faster than dereferencing nodes in a tree structure in Java. This implementation of binary heap does not include the $decreaseKey$ operation, since 1) $decreaseKey$ operation is expensive, 2) $decreaseKey$ operations are not a must, since every time we it is necessary to decrease the key of a vertex in the heap, a new element pointing to the same vertex can be inserted instead, with the decreased key. Then the newly inserted copy should be extracted earlier than the old copy and after the vertex is closed, the old copy, which also points to the same vertex, will be skipped. Using this scheme, it is necessary to protect the binary heap from overflowing, since a vertex can be inserted multiple times, although in sparse graphs overflow is rare if the initial size is set to be the number of vertices. A way to get around this is to make the binary heap a dynamically extensible array. That is, whenever an overflow is about to occur, the size of array is extended by a factor of λ ($\lambda > 1$). In our implementation, λ is set to be 2. The

enqueue method which did the automatic extension is listed as follows.

```
/**
 * Insert an entry to the bindary heap.
 * If the bindary heap is full, it will automatically double its
 * size.
 */
public void enqueue (Comparable object)
{
    if (count == array.length - 1)
    {
        // BinaryHeap is full, doubling it.
        Comparable[] newArray = new Comparable[array.length<<1];
        System.arraycopy(array,0,newArray,0,array.length);
        array = newArray;
    }
    ++count;
    int i = count, parent;
    while (i > 1 && array[parent=(i>>1)].compareTo(object)>0)
    {
        array[i] = array[parent];
        i = parent;
    }
    array[i] = object;
}
```

In this code, "array" is the array containing elements in the binary heap; "count" is the actual number of elements in the array. The first if-statement checks the array size to do an extension if necessary. The while-loop does the actual job of inserting the element to the binary heap.

We do not do array-shrink when an element is extracted from the binary heap since 1) Array copy is an expensive operation. It should be avoided unless it is necessary. 2) Many small-sized array allocated frequently could make the memory fragmented resulting in the slowing down of Java's garbage collector.

Details in U-Heap

Since a U-Heap is a full and complete binary tree as defined in Section 3.2.1, it is very easy to implement it using an array. To construct a U-Heap for boundary sets,

it is necessary to know the number of boundary sets l first. Each of the boundary sets corresponds to a leaf node in the U-Heap. According to the nature of full and complete binary tree, the totally number of nodes in the U-Heap is $2l - 1$, which can be easily get from mathematical induction. The code for initializing a U-Heap is as follows:

```
/**
 * Initialize a U-Heap according to an input array d.
 * Input: d - data array. The elements should be the leaf nodes of
 * the U-Heap.
 */
public UHeap(KeyValuePair[] d)
    {
        // allocate an array for the U-Heap,
        // the size of the array is 2 * d.length - 1
        size = d.length;
        data = new KeyValuePair[(size<<1)-1];

        // build up leaf nodes
        System.arraycopy(d,0,data,size-1,size);
        Arrays.sort(data,size-1,(size<<1)-1); // sort the data set by
                                              // their keys

        // build the heap structure by their "value" -- fill out
        // non-leaf nodes
        KeyValuePair left, right;
        Comparable l,r;
        for ( int i = size - 2; i >= 0; i-- )
        {
            left  = data[(i<<1)+1];
            right = data[(i<<1)+2];
            l = (Comparable) left.getValue();
            r = (Comparable) right.getValue();
            data[i] = (l.compareTo(r) > 0) ? right : left;
        }
    }
```

Fibonacci Heap and Distance Vector

A Fibonacci heap is a collection of heap-ordered trees. To be specific, it is a collection of "unordered" binomial trees [1]. Our implementation of the Fibonacci

heap is based on the Chapter 22 of [1]. More details on how to implement the Fibonacci heap can be found there. Here we only introduce the usage of the Fibonacci heaps in the distance vector data structure, which keeps the boundary vertices in a boundary set.

Fibonacci heaps are used for organizing boundary vertices in distance vectors in "heap order". The key for a boundary vertex is its shortest distance from the source vertex s. Since the Fibonacci heap does not provide "search" functionality, operations (such as $decreaseKey$) which refer to a given Fibonacci node, require a pointer to that node as part of their input. Therefore, when the Fibonacci heap is constructed, all its newly generated nodes should be kept in a dictionary or map. In this implementation, there are two hash maps in the distance vector as well as the Fibonacci heap. The class variables and constructors of the DistVect class is as follows:

```
public class DistVect implements Serializable
{
    HashMap table;     // (key=Coordinate,value=BsAux) pair
    FibHeap queue;     // open vertices in heap order by their
                       // distances,
    HashMap openVertices; // contains the mapping from the open
                          // vertices the FibHeapNodes

    /**
     * Construct a distance vector.
     * @param boundaryVertices the boundary vertices of the
     * boundary set.
     */
    public DistVect(Coordinate[] boundaryVertices)
    {
        int n = boundaryVertices.length;
        table = new HashMap();
        openVertices = new HashMap(n);
        queue = new FibHeap();
        FibHeapNode fhn;
        for ( int i = 0; i < n; i ++ )
        {
```

```
            table.put(boundaryVertices[i],new BsAux());
            fhn = queue.insert(boundaryVertices[i],Long.MAX_VALUE);
            openVertices.put(boundaryVertices[i],fhn);
        }
    } // End of DistVect()
}    // End of Class DistVect
```

In this implementation, "queue" is a Fibonacci heap. All nodes in the heap are initialized to Long.MAX_VALUE. The hash map "table" contains a mapping from a boundary vertex to its shortest path information kept in BsAux. This information includes the shortest distance from s so far, the parent vertex in to shortest path, and whether it is closed or not. The second hash map "openVertices" contains a mapping from an open boundary vertex to a Fibonacci node in the Fibonacci heap. Therefore, whenever a Fibonacci heap operation needs a node as a parameter (for example $decreaseKey$ to a boundary vertex), the "openVertices" can be looked up by giving the coordinate of the boundary vertex.

Details in Distance Matrix

A distance matrix could be simply a 2-dimensional matrix that contains the shortest distances between every pair of boundary vertices in a fragment. However, since the goal is to retrieve the shortest distance by two boundary vertices, it is necessary to keep the boundary vertices in the distance matrix too. A simple way to do this is to keep the boundary vertices sorted in an array; the index of the boundary vertex in the vertex array is the index in the distance 2-D array. For example, suppose the indices of boundary vertex u and v are i and j respectively, the shortest distance can be found in the distance array $[i, j]$. Another trick that can be played for an undirected graph is based on the symmetric property of the distance array. More

than 50% space can be saved by only recording the shortest distance from i to j where $i < j$. Then the 2-D distance matrix can be represented by a 1-D array, where rows in the upper triangle matrix are appended to the array head by tail. The length of the array is given by $n(n-1)/2$ where n is the number of boundary vertices in the fragment. Given two indices in the vertex array i and j, the shortest distance between them are given by:

$$\text{shortestdistance} = \begin{cases} 0 & i = j \\ \text{matrix}[i*(2n-i-1)/2+j-i-1] & i < j \\ \text{matrix}[j*(2n-j-1)/2+i-j-1] & i > j \end{cases}$$

The code for DistMatrix constructor and get method is given as follows:

```
public class DistMatrix implements Serializable
{
    Coordinate[] coords; // ordered coordinates of boundary vertices
    long[] linearMatrix; // 1-D array simulating distance matrix
    int n;

    /**
    /* Construct the DistMatrix object.
    /* @param c the boundary vertices.
    /* @m the distance matrix
    */
    public DistMatrix(Node[] c, long[][] m)
    {
        coords = new Coordinate[c.length];
        for ( int i = 0; i < c.length; i++ )
            coords[i] = c[i].getCoordinate();
        int index = 0,len;
        n = m.length;
        linearMatrix = new long[n*(n-1)/2];

        for ( int i = 0; i < n; i++ )
        {
            len = n-i-1;
            System.arraycopy(m[i],i+1,linearMatrix,index,len);
            index += len;
        }
```

```
    }

    /**
    /* Get the shortest distance between two vertices with indices
     * i and j.
    /* @param i the index of vertex 1.
    /* @param j the index of vertex 2.
    /* @return the shortest distance between vertex 1 and 2.
    */
    public long get(int i, int j)
    {
        if ( i == j )
            return 0;
        if ( i < j )
            return linearMatrix[(i*((n<<1)-i-1)>>1)+(j-i-1)];
        else // (j < i)
            return linearMatrix[(j*((n<<1)-j-1)>>1)+(i-j-1)];
    }
 ...
}
```

Chapter 5

Experiments

In chapter 3 and chapter 4, we have seen many data structures and heuristics designed to optimize the running time in terms of both CPU and I/O time. In this chapter, we will see the experimental results on real-world digital maps. We test the correctness, efficiency, and effectiveness of my implementation. The testing is divided into two parts: the pre-processing phase and the querying phase. In the pre-processing phase, the most important thing is the correctness. Although the running time is also optimized for certain algorithms (such as in the k-pair shortest paths algorithm), we do not focus on the performance issue as long as it can be done in a reasonable time (it may take days for very large digital maps). On the other hand, the running time is the most important measurement in the querying phase, so we want to focus on the efficiency of the pruning algorithm and disk-based shortest path algorithm in the second phase.

The computer for testing is a dual-processor system with two Pentium III 933MHz CPU's and 1 GB SDRAM, with 16KB level 1 cache and 256KB level 2 cache. The hard disk is Ultra 160 SCSI drive. The operating system is Microsoft

Windows 2000 Server SP1. We use Sun Java 1.3 HotSpot Client VM (build 1.3.0-C mixed mode), and the relational database is IBM DB2 Universal Database Server 7.0.

The programs can be run on less powerful computers (for example, a Pentium II 333MHz with 128MB memory and EIDE hard drive), as long as the minimum memory requirement (depending on the maximum fragment size you specified when partitioing) is met. The only difference is the running time, which depends largely on the power of CPU and the capacity and bandwidth of main memory.

5.1 Pre-processing phase

In the pre-processing phase, there are five steps: creating TIGER database in a relational database system, converting TIGER database to Abstract Line Features (ALF), generating object data files and Hilbert R-tree for the geometric objects, partitioning the object file into fragments using Hilbert R-tree, and calculating k-pair shortest paths between every pair of boundary vertices in each fragment. In these five steps, I will not analysis the performance of creating Hilbert R-tree on geometric objects because it was evaluated by other papers in our project group [48].

5.1.1 Creating Tables for TIGER/Line Data Source

The TIGER/Line data is stored in plain text files. There are seventeen files corresponding to seventeen records for each county. In the relational database, we need to create one table for each record. Data in the same record type in different counties are put in the same table. We use Java Data Base Connection (JDBC)

as interface for accessing IBM DB2 database. The experimental results show that the running time is proportional to the number of tuples inserted into the tables. Reading from the TIGER/Line files only contributes little to the running time. On average, inserting every 1,000 tuples takes 6.33 seconds, while reading 1,000 lines of TIGER/Line records and parsing it only takes about 0.013 second. Figure 5.1 shows the relationship between the running time and the number of tuples inserted into the tables.

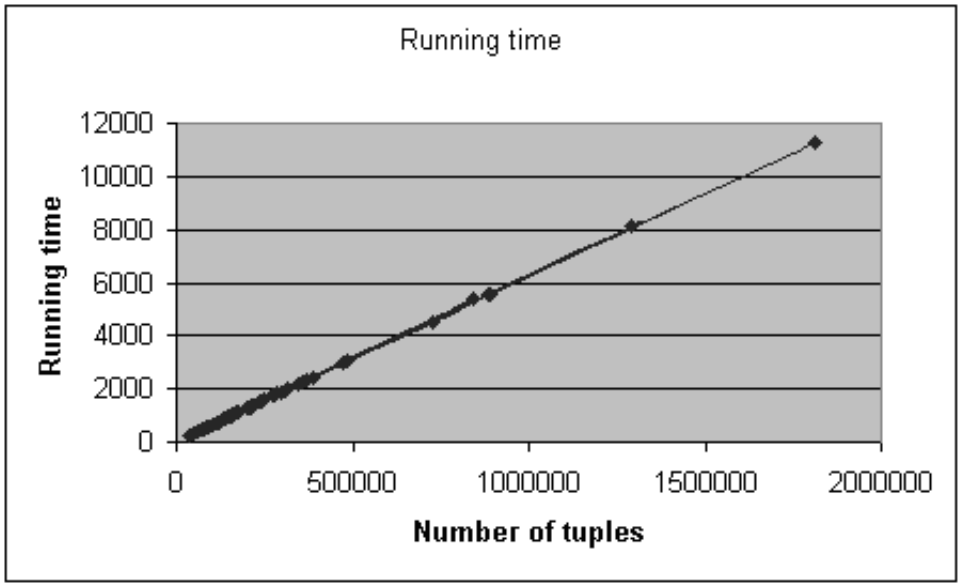

Figure 5.1: Running time of populating TIGER Databases

5.1.2 Creating Tables for Abstract Line Features

When creating tables for Abstract Line Features (ALF), we need to read tuples from TIGER database, process the data, and save the results to ALF tables. Therefore, unlike creating TIGER database, we need to measure the time for querying as well

as insertion in RDBMS. Again, the major part (97% in average) of running time is due to the execution of the SQL statements - queries and insertions in RDBMS. Data processing is minimized to just a few simple type conversions for each tuple. The SQL query statements are very simple like: "SELECT tlid, cfcc, frlong, frlat, tolong, tolat, side1, countyl, countyr, statel, stater FROM tgr06001rt1 WHERE cfcc LIKE 'A_'". They do not have subqueries and aggregations. Joins are simply between two tables. Insertion is also simple insertion without subqueries. In these two types of SQL statements, insertions take about 91% of SQL time (i.e. 89% of total running time). The percentages of running time contributed by different operations are shown in the Figure 5.2.

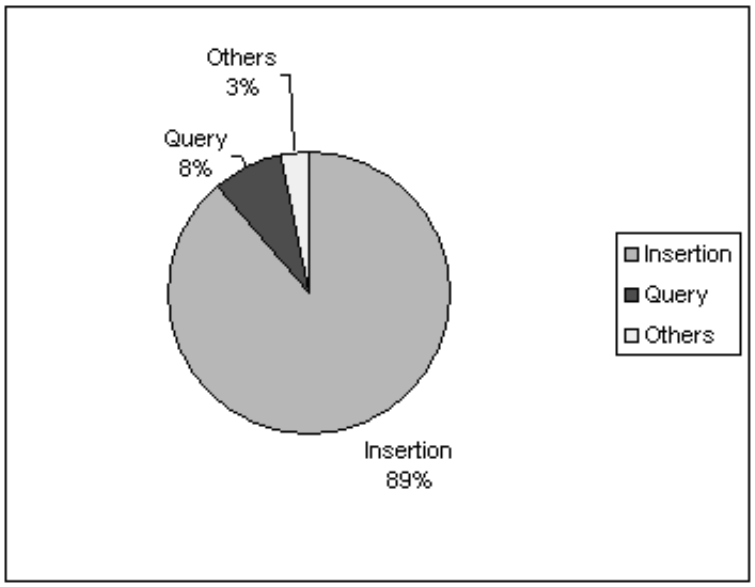

Figure 5.2: Time Distributtion of Making Abstract Line Features

According to the pie diagram of Figure 5.2, there is little to optimize since insertions in most RDBMS are usually not optimized for running time performance.

Rather, much overhead done for ensuring the data integrity and consistency slows down the insertion process.

5.1.3 Partitioning Digital Maps into Fragments

In this experiment, we set the minimum and maximum number of vertices in fragments to be 15,000 and 20,000. Road systems of seven states are chosen to form four testing digital maps: Connecticut (CT), New Mexico (NM), California (CA) and the eastern five states (East5, which is composed of Connecticut, Massachusetts, New Jersey, New York, and Pennsylvania). The running time is divided into three measurements: I/O time of virtual data structures, Hilbert R-Tree querying time, and others as shown in Figure 5.3.

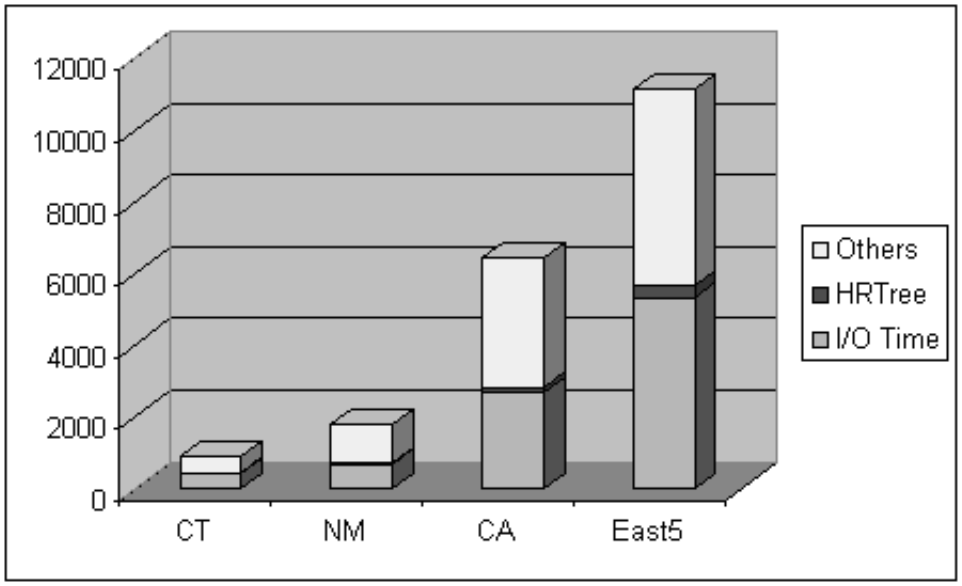

Figure 5.3: Time Distribution of Partitioning Algorithm on Different Size Data

The time spent other than I/O and Hilbert R-Tree is mainly due to BFS graph

traversing and finding the boundary vertices between every pair of fragments. According to Figure 5.3, this process together with I/O time for virtual data structures constitute the major part of total running time. The relationship between the total running time and the number of arcs in the digital map is shown in Figure 5.4. It is clear that the total running time is almost linear to the number of edges, which is optimal for partitioning large digital maps.

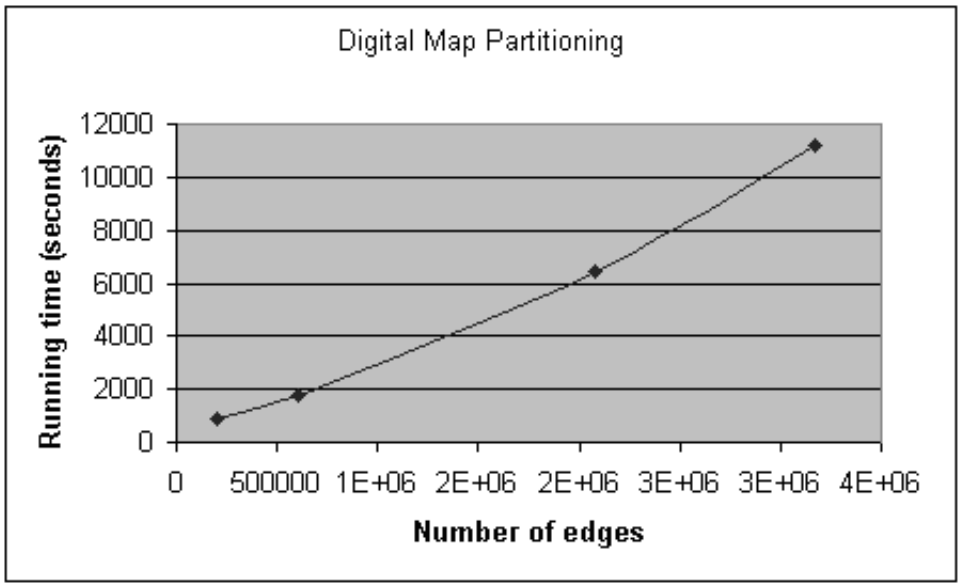

Figure 5.4: Running time of comparing with number of edges

The statistic values (number of boundary sets in each fragment, number of boundary vertices in each fragment, and so on) are shown in Table 5.1.

From the table, we can see that the average numbers of boundary vertices in boundary sets are steadily around 40 to 50. The average number of boundary sets per fragment is from 1.8 to 2.45. The average number of arcs in fragments is from 19,045 to 19,952. The relation between number of arcs and number of boundary

State (Number of Arcs)	Number of fragments	Number of boundary sets	Number of boundary vertices	Average number of boundary vertices per boundary set
CT (199518)	10	18	1003	55
NM (609424)	32	76	3216	42
CA (2079668)	108	265	11705	44
East5 (3169730)	164	402	19434	48

Table 5.1: Partitioning results: fragments, boundary sets, and boundary vertices

vertices is shown in Figure 5.5.

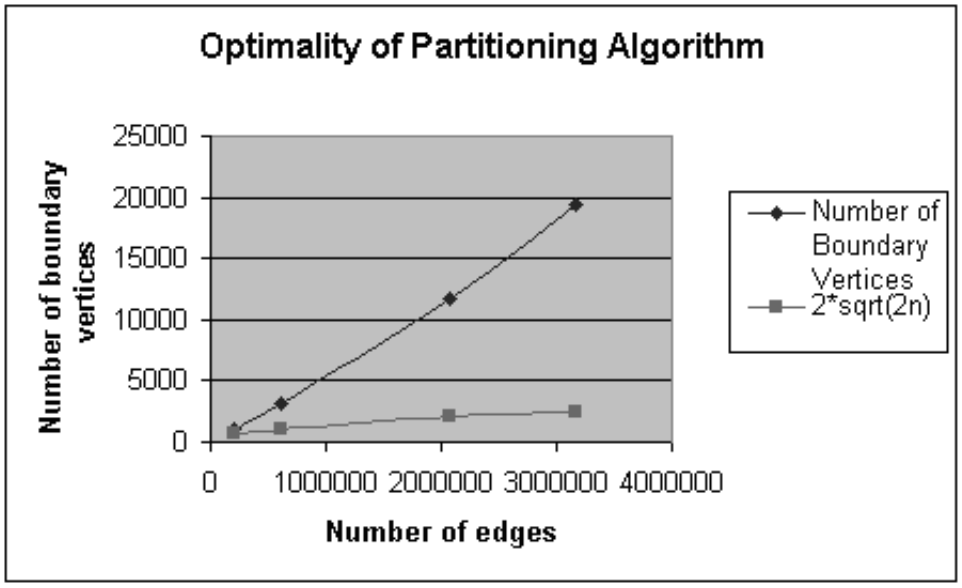

Figure 5.5: Relation between number of edges and number of boundary vertices

From the experimental results, our partitioning algorithm is not as optimal (in terms of number of boundary vertices) as Lipton and Tarjan's $2\sqrt{2n}$ planar separator in planar graphs. Rather, the number of boundary vertices looks like in

$O(n)$. If we know the input graph is planar beforehand, implementing a disk-based Lipton and Tarjan's planar graph separator algorithm may give a much better result for the disk-based shortest path algorithm.

5.1.4 Calculating k-pair Shortest Paths

The last step in the pre-processing phase is to compute the k-pair shortest paths and materialize the results (distance matrices and sketch graph) on hard disk. The running time can be divided into two parts: I/O time and k-pair shortest path calculating time, in which the latter occupies the major part. The statistics recorded in the four test cases are shown in the Figure 5.6.

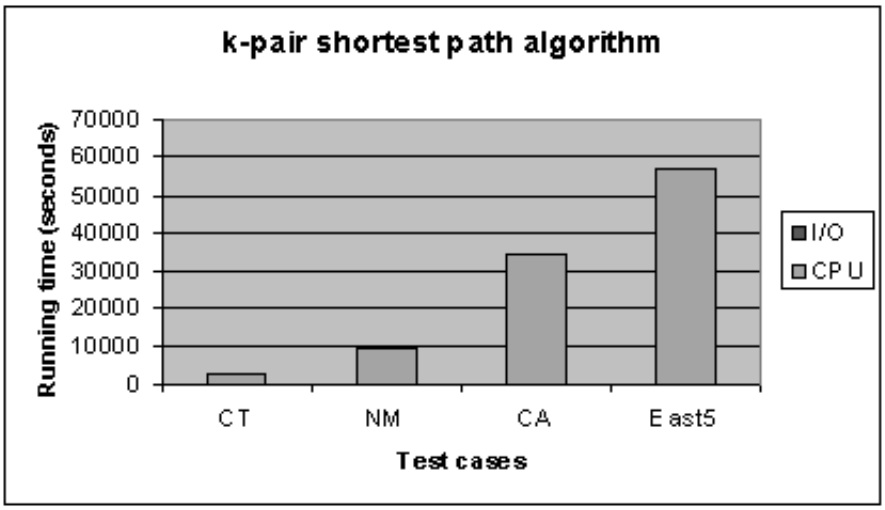

Figure 5.6: Running Time of k-pair Shortest Path Algorithm

The relationship of running time and number of edges and number of boundary vertices is shown in Figure 5.7.

From this diagram, we can see that the running time of k-pair shortest path

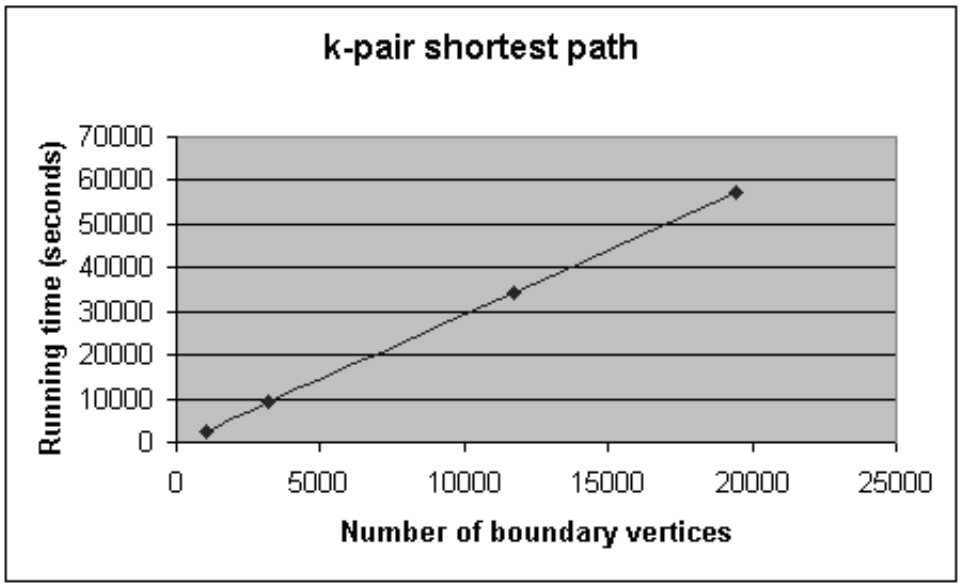

Figure 5.7: Relation between running time and boundary vertices

algorithm is almost linear to the number of boundary vertices. It is better than the Johnson's all-pair shortest path algorithm, which is in $O(n^2 \log n)$. If the number of boundary vertices can be decreased to $O(\sqrt{n})$, the running time can be reduced greatly, which is very good for extending the algorithms to dynamic graphs.

5.2 Querying Phase

In the query phase, there are two steps: pruning the sketch graph given the source and destination vertices, and finding the shortest path between them using disk-based shortest path algorithm.

5.2.1 Pruning Sketch Graph

For the pruning algorithm, we test the efficiency as well as its effectiveness. That is, we have to test how long it takes to prune the sketch graph and how much computation (CPU and I/O time) can be saved due to the pruning. The time the pruning algorithm takes is composed of two parts: building the shortest path trees in the source and destination fragments, and checking the α and β values of all super edges to prune the super nodes. In these two parts, the second part can be considered solely as the cost of pruning process. As for the first part, the result can be shared by the disk-based shortest path algorithm. Therefore, when we talk about the cost of pruning algorithm, we only consider the second part (denoted by T_c). The saving of pruning algorithm also consists of two parts: the time saving because of fewer distance vectors being constructed and fewer nodes in U-Heap being generated (denoted by T_1), and the time saving because of fewer main thrust operations being performed (denoted by T_2). T_1 is determined by the number of super nodes pruned, and T_2 is determined by the number of main thrust operations saved. The effectiveness of the pruning algorithm is measured by the benefit of pruning, which is defined as the actual saved running time ($T_1 + T_2 - T_c$). If the value is negative, then the saving in shortest path algorithm is not worth the cost of pruning process.

We tested the program on the digital map of New Mexico State with 1,000 randomly generated test cases. The 1,000 test cases are divided into three categories based on the geographical relation between the source fragment S and destination fragment D: non-adjacent, adjacent, and equal (i.e. $S = D$). Table 5.2 shows the

results.

S and D are	Number of test cases	Total number of SuperNodes removed	Total number of MainThrust() saved	Pruning time (seconds)
Non-adjacent	838	1,759	106	354.267
Adjacent	140	4,291	432	53.254
Equal	22	1,256	243	8.423

Table 5.2: Testing Results of Pruning Algorithm

From table, we can see that the cost of pruning algorithm (pruning time) is approximately proportional to the number of test cases. It means that pruning cost is independent of the geographical relationship between S and D. (This actually makes sense because according to the theoretical complexity analysis in Section 5.2, the cost of pruning algorithm should only depend on the size of the sketch graph). On the other hand, the saving of pruning process does depend on the geographical relationship between S and D. In the 1,000 randomly generated test cases, most of them (83.8%) are in non-adjacent category, but they only contribute 24.1% and 13.6% of the total savings in ΣT_1 (total time savings because of less distance vectors and nodes in U-Heap) and ΣT_2 (total time savings for less MainThrust method calls) respectively. The average benefit for one test case is -0.41 seconds (based on our experiments, the average time spent on constructing a distance vector and a node in the U-Heap corresponding to a super node in the sketch graph is 0.00267 second. The average time spent on one MainThrust method is 0.029 second). Therefore, in general, it is not worth pruning if the two vertices are in non-adjacent fragments. In contrast, only 2.2% of the test cases are in the category of "S equals to D" as shown in Table 5.2, but they contribute to 17.2% and 31.1% of ΣT_1 and ΣT_2 respectively.

The average benefit per test case is 0.09 second. The rest of the saving are due to the test cases that S and D are adjacent fragments, whose average benefit is -0.21 second. From the experimental results, we can see that pruning algorithm is not very effective when the source and destination vertices are far apart. The pruning process is worthwhile only when the two vertices are in the same fragment, but the benefit is insignificant (much less than 1 second on average).

5.2.2 Disk-based Shortest Path

For the disk-based shortest path algorithm, we tested the correctness of the algorithm and its running time performance. For correctness, two ways were taken: boundary case testing and random testing. The result was compared with the shortest path got from main memory version of Dijkstra's algorithm.

Correctness Testing

We came up with seventeen boundary test cases shown below to ensure the special cases are handled correctly. Suppose the source vertex is s and the destination vertex is d.

1. Shortest path passed a boundary vertex that is in multiple boundary sets.
2. s and d are in two different connected components.
3. s and d are in the same fragments.
4. s and d are in adjacent fragments.
5. s and d are in non-adjacent fragments.
6. s and d are in the same fragment, and s is a boundary vertex.
7. s and d are in the same fragment, and d is a boundary vertex.
8. s and d are in the same fragment, and s and d are both boundary vertices.

9. s and d are in adjacent fragments, and s is a boundary vertex.
10. s and d are in adjacent fragments, and d is a boundary vertex.
11. s and d are in adjacent fragments, and s and d are both boundary vertices.
12. s and d are in non-adjacent fragments, and s is a boundary vertex.
13. s and d are in non-adjacent fragments, and d is a boundary vertex.
14. s and d are in non-adjacent fragments, and s and d both are boundary vertices.
15. $s = d$, and s is not boundary vertex.
16. $s = d$, and s is a boundary vertex.
17. s and d are boundary vertices and in the same boundary set.

In addition to these boundary test cases, we also tested it by randomly generate pairs of source and destination. The number of randomly generated test cases is 1,000 for both Connecticut and New Mexico states. All results equal to the result got from main memory version of Dijkstra's algorithm.

Performance Testing

For performance testing, we examine the effects of different parameters to the running time. By different parameters, we mean the number of entries in the cache for virtual data structures, and the amount of main memory allocated to the Java Virtual Machine when it starts up. Among the virtual data structures, distance vector (DistVect) and distance matrix (DistMatrix) are the two most frequently used virtual data structures (both are virtual hash tables). To be more accurate, we do not count the running time of FillSP algorithm, since that depends on how far apart the destination from the source vertex. To simplify the experiment, we first see the effects of the cache sizes of distance matrix and distance vector to the running time when the amount of memory is sufficient for Java Virtual Machine

to put the maximum number of objects in the virtual data structures, then we see what if the memory is not enough to fill in the caches in virtual data structures. Figure 5.8 shows the relation among the size of caches for distance vectors and distance matrices and the running time using 500MB memory for JVM.

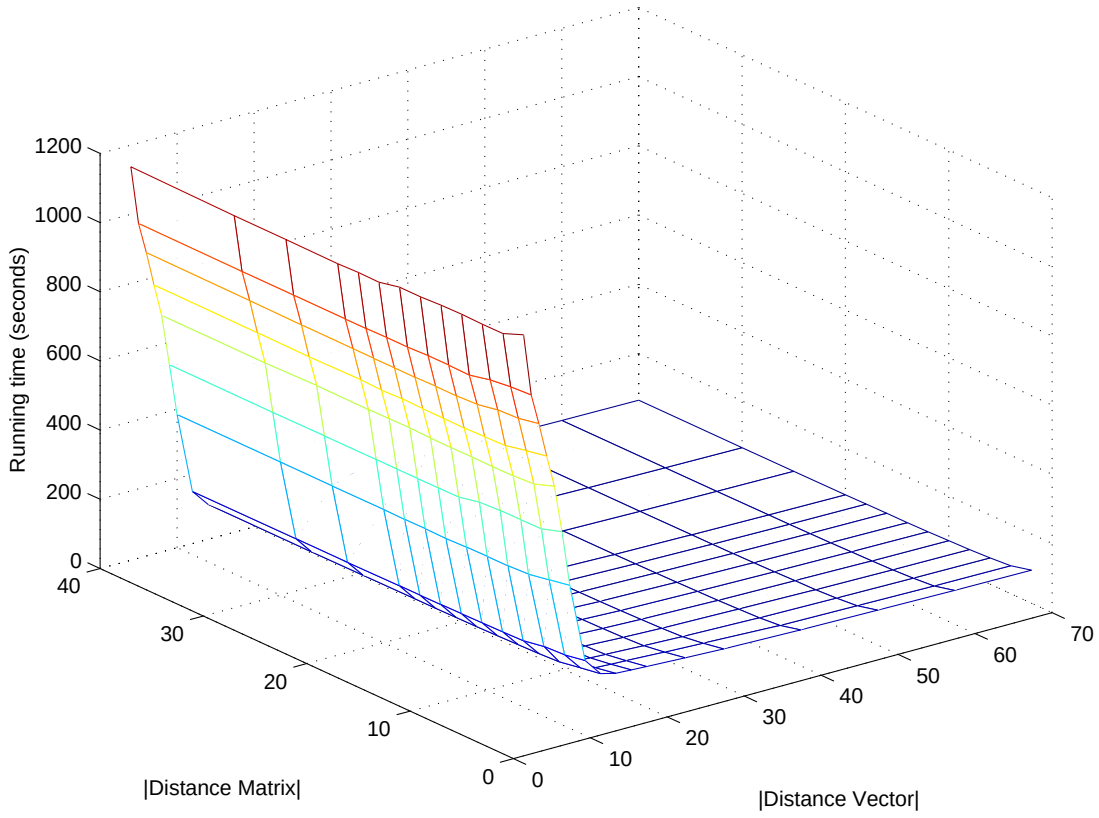

Figure 5.8: Running time with various sizes of distance vectors and distance matrices with 500MB for JVM

From the figure, we can see that the cache size for distance vector greatly affect the running time if it is less than 18, but the cache size of distance matrices does not affect too much of the performance. This can be seen from the cross section of the figure along $|DistanceVector|$ axis and $|DistanceMatrix|$ axis, which is shown

in Figure 5.9.

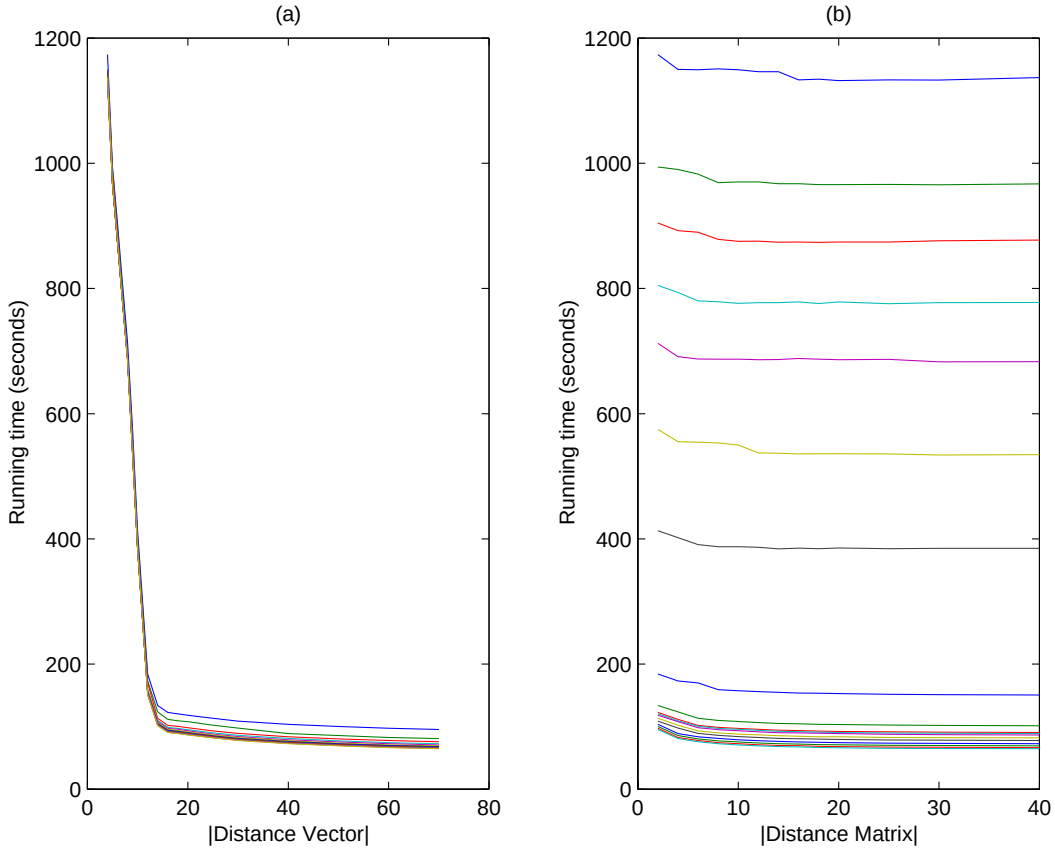

Figure 5.9: The effects of cache size of distance vector and distance matrix (a) running time for different cache size of distance vectors (b) running time for different cache size of distance matrix

Another parameter affecting the performance is the amount of memory available for JVM. Figure 5.10 shows the running time for different cache sizes of distance vector and distance matrix using 60MB memory for JVM.

From the figure, we can see that different amount of memory for JVM does not change the "shape" of the surface. This is so because JVM garbage collector does not bias to distance vector or distance matrix. Therefore, the only result is that

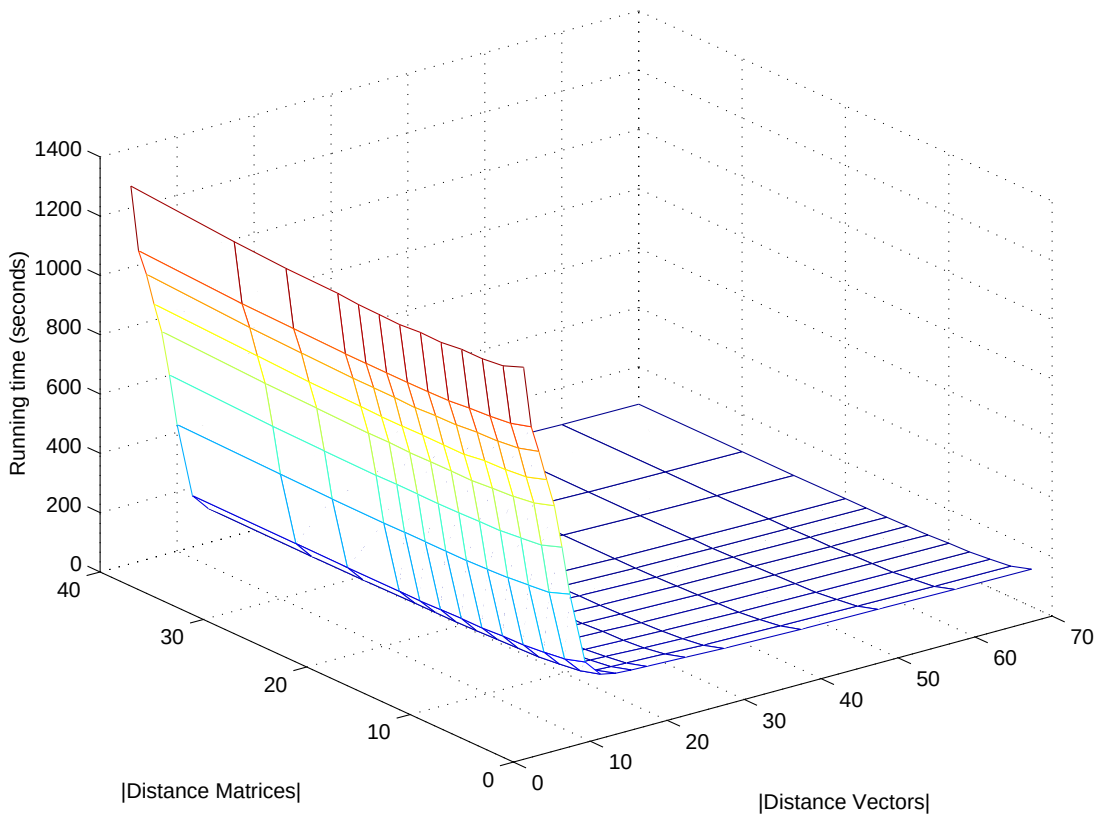

Figure 5.10: Running time with various sizes of distance vectors and distance matrices with 60MB for JVM

running time is decreasing when more memory is allocated to JVM. Figure 5.11 shows the running time for different amount of memory assigned to JVM when the cache size of distance vector equals to 4 and the cache size of distance matrix equals to 2.

Another thing we want to test is the difference between binary heap and Fibonacci heap in shortest path algorithms for sparse graphs and dense graphs. We can think of road systems as sparse graphs since they are usually planar, whereas super graphs can be thought of as dense graphs since they are composed of cliques.

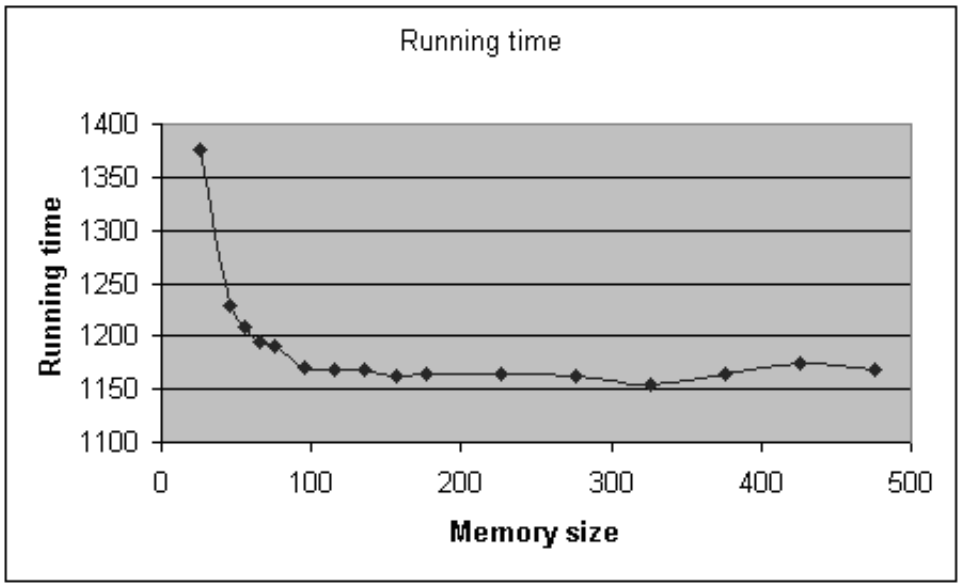

Figure 5.11: Running time for different memory size with $|DistMatrix| = 2$, $|DistVector| = 4$

We test the running time of disk-based shortest path algorithm using binary heaps and Fibonacci heaps respectively for containing the vertices in distance vectors. The results show that the average running time using Fibonacci heaps is 10% faster than using binary heaps. The reason for that is because there are many decreaseKey operations during MainThrust method, and Fibonacci heap has a less asymptotic complexity for this operation than binary heap. However, when the Fibonacci heap is used for Dijkstra's shortest path algorithm in fragments (sparse planar graphs), the performance is 2% slower than using binary heap. The reason for that is that Fibonacci heap's relatively large constant outweighs the relatively small number of decreaseKey operations.

Chapter 6

Conclusions and Future Work

6.1 Summary

Memory-based shortest path algorithms are very well studied and many theoretical and empirical results have come out. However, disk-based or external memory shortest path algorithms are not studied very well. Some of the previous works have built some theoretical models and have gotten empirical results from prototypes, but none of them has shown that their algorithms are practical to very large spatial databases such as the road system of California or even larger digital maps. My thesis proposed a disk-based algorithm working like Dijkstra's algorithm and had gotten promising empirical results from the real road systems. Experiment shows that the running time of our disk-based shortest path algorithms is about two to four times slower than the main memory version of Dijkstra's algorithm given that the memory is large enough. This conclusion is based on the assumption that the whole map can be fit into main memory and it is already loaded in advance. If the I/O time for loading the whole graph is counted, our disk-based shortest path algorithm is even faster in most cases. This is because we make use of the

pre-computation information and it is well clustered on the disk.

An important factor of the efficiency of our algorithm is the data structure we used for the priority queue. Fibonacci heaps are better than binary heaps when the graph is dense but are not as good when the graph is sparse. Fortunately, we can differentiate dense graphs from sparse graphs offline, so applying different priority queues to different kinds of graphs is possible.

Another even more important parameter is the cache size of virtual data structures. Experiments show that if the cache size is less than a certain value for distance vector, the performance deteriorates very fast. This is because of the locality of relaxation operations in the shortest path algorithm.

6.2 Future Works

In this thesis, we have shown that it is possible to answer the shortest path queries in very large spatial databases quite fast. However, in order to make it applicable to answer hundreds of queries per seconds in near real-time, much work has to be done. Here I propose the most important and obvious work that should be done in the future.

- Parallel computing: the idea behind our algorithm is divide-and-conquer, for which parallel computing is one of the most suitable ways to speed up the query processing phase. In the disk-base shortest path algorithm, when a boundary vertex is the next closed vertex, all boundary sets in the same fragment with the boundary vertex should be relaxed. This process can be done in parallel, i.e. one thread is responsible for relaxing one boundary

set. Since the MainThrust is the most costly operation in the shortest path algorithm, this parallelism can contribute a lot to the overall performance.

- Different replacement algorithms for virtual data structures: right now we are using the Least Recently Used (LRU) algorithm to determine which item in the cache should be swapped out to the hard disk. For shortest path problem, it may not be the most appropriate algorithm. In the future work, modifications need to be done to the virtual data structure to let the programmer choose replacement algorithms easily.

- Different priority queues for sparse and dense graph shortest path algorithms: we have tested the two most commonly used priority queues - binary heap and Fibonacci heap. There are other data structures which may be more efficient than these two in our particular applications.

- Lipton and Tarjan's planar graph separator for planar graphs: if the digital map is planar, Lipton and Tarjan's planar graph separator algorithm could result in much fewer boundary vertices. This can greatly reduce the computation time of k-pair shortest path algorithm and the disk-based shortest path algorithm as well.

- Dealing with dynamic graphs and online shortest path algorithms: in the real world, the weight of an edge could be more complex than just the length of the street block. It can be a function of time, or something you do not know beforehand. Therefore, how to deal with the dynamic graph and online problems is a very important and realistic problem. Some work has been done

in theory, but not much in practice, especially for very large spatial databases. This problem remains a major challenge for the future research.

Bibliography

[1] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest, *Introduction to Algorithms*, MIT Press, 1990.

[2] Stuart J. Russell, Peter Norvig, *Artificial Intelligence: A Modern Approach*, Pretence Hall, 1994.

[3] F. Harary, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[4] Rober Endre Tarjan, *Data Structures and Network Algorithms*, Society for Industrial and Applied Mathematics, 1983.

[5] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company, 1974.

[6] Gilles Brassard, Paul Bratley, *Fundamentals of Algorithms*, Prentice Hall Inc., 1996

[7] Robert G. Busacker, and Thomas L. Saaty, *Finite Graphs and Networks: An Introduction with Applications*, McGraw-Hill Inc. 1965.

[8] Narsingh Deo, and Chi-yin Pang, *Shortest Path Algorithms: Taxonomy and Annotation*, Networks 14 pp. 275–323, 1984.

[9] Joseph S. B. Mitchell, *Geometric Shortest Paths and Network Optimization*, Handbook of Computational Geometry, Elsevier Science, Amsterdam, 1998.

[10] Ning Jing, Yun-wu Huang and Elke A. Rundensteiner, *Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its performance Evaluation* , IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 3 May/June 1998.

[11] Yun-wu Huang, Ning Jing, and Elke A. Rundensteiner, *Efficient Graph Clustering for Path Queries in Digital Map Databases*, Proceeding of the 5th International Conference on Information and Knowledge Management, 1996.

[12] Shashi Shekhar, Andrew Fetterer, and Brajesh Goyal, *Materialization Trade-Offs in Hierarchical Shortest Path Algorithms*, Proceeding of the International Symposium on Large Spatial Databases, Springer Verlag (Lecture Nodes in Computer Science), 1997.

[13] Shashi Shekhar, Ashim Kohli, and Mark Coyle, *Path Computation Algorithms for Advanced Traveler Information Systems (ATIS)*, IEEE 9th International Conference on Data Engineering, pp 31-39, 1993.

[14] Ning Jing, Yun-Wu Huang, and Eike Rundenstener, *Hierarchical Optimization of Optimal Path Finding for Transportation Applications*, Proceeding of the Conference on Information and Knowledge Management, 1996. pp. 261-268.

[15] B. V. Cherkassky, A. V. Goldberg, T. Radzik, *Shortest Path Algorithms: The-*

ory and Experimental Evaluation, Mathematical Programming, Vol. 73, 129-174, June 1996.

[16] David Eppstein, *Finding the k-shortest Paths*, SIAM J. COMPUT. Vol. 28, No.2, pp. 652-673, 1998.

[17] Monika R. Henzinger, Philip Klein, Satish Rao, and Sairam Subramanian, *Faster Shortest Path Algorithms for Planar Graphs*, Proceeding of the 26th ACM Symposium on Theory of Computing, 1994, pp. 27-37.

[18] Giorgio Ausiello, Giuseppe F. Itatiano, Alberto Marchetti Spaccamela, and Umberto Nanni, *Incremental Algorithms for Minimal Length Paths*, Journal of Algorithms, 12 (1991), pp. 615-638.

[19] Andrew V. Goldberg, Robert E. Tarjan, *Expected Performance of Dijkstra's Shortest Path Algorithm*, Technical Report 96–062, NEC Research Institute, Princeton, NJ, 1996.

[20] Andrew V. Goldberg, Craig Silverstein, *Computational Evaluation of Hot Queues*, Technical Report 97–104, NEC Research Institute,June, 1997.

[21] Boris V. Cherkassky, Andrew Goldberg, and Craig Silverstein, *Heap-on-Top Priority Queues*, Technical Report 96-042, NEC Research Institute, Princeton, NJ, 1996.

[22] Andrew Goldberg, Craig Silverstein, *Buckets, Heaps, Lists, and Monotone Priority Queues*, In Proceedings of the 8th ACM-SIAM Symposium on Discrete Algorithms, pp. 83-92, 1997.

[23] Robert Tarjan, *Solving Path Problems on Directed Graphs*, STAN-CS-75-528, Nov. 1975, CS Department, Stanford University.

[24] John Hopcroft, and Robert Tarjan, *Efficient Algorithms for Graph Manipulation*, STAN-CS-71-207, March 1971, CS Department, Stanford University.

[25] Divide R. Karger, Daphne Koller, Steven J. Phillips, *Finding the Hidden Path: Time Bounds for All-Pair Shortest Paths*, SIAM Journal of Computing, 22(6): 1199-1217, 1993.

[26] Greg N. Frederickson, and Ravi Janardan, Designing Networks with Compact Routing Tables, Algorithmica 3: 171-190, 1988.

[27] Brenda S. Baker, *Approximation Algorithms for NP-Complete Problems on Planar Graphs*, Journal of ACM 41(1), 153-180, 1994.

[28] Alan George, *Nested Dissection of Regular Finite Element Mesh*, SIAM Journal of Numerical Analysis, Vol. 10 No2, April 1973.

[29] George Karypis, and Vipin Kumar, *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs*, Technical Report TR 95-035, Department of Computer Science, University of Minnesota, Minnesota, 1995.

[30] George Karpis, and Vipin Kumar, *Analysis of Multilevel Graph Partitioning*, Technical Report TR 95-037, Department of Computer Science, University of Minnesota, Minnesota, 1995.

[31] Greg N. Frederickson, *Planar Graph Decomposition and All Pair Shortest Paths*, Journal of ACM 38(1): pp. 162-204, 1991.

[32] Greg N. Frederickson, *Searching Among Intervals and Compact Routing Tables*, Algorithmica 15(5): pp. 448-466, 1996.

[33] Greg N. Frederickson, *Using Cellular Graph Embedding in Solving All Pair Shortest Path Problems*, Journal of Algorithms 19(1), pp. 45-85, 1995.

[34] Richard J. Lipton, and Robert Endre Tarjan, *A separator Theorem for Planar Graphs*, SIAM Journal Applied Mathematics, 36 (1979) pp. 177-189.

[35] Greg. N. Frederickson, *Fast Algorithms for Shortest Paths in Planar Graph, with Application*, SIAM Journal of Computing, Vol. 16, No. 6, December 1987.

[36] Shimon Even, and Hillel Gazit, *Updating Distances in Dynamic Graphs*, Methods of Operations Research, 49(1985), 371-387.

[37] Daniele Frigioni, Mario Ioffreda, Umberto Nann, and Giulio Pasqualone, *Experimental Analysis of Dynamic Algorithms for Single Source Shortest Path Problems*, In Proceedings of the 1st Workshop on Algorithm Engineering (WAE), September 11-13, 1997, Venice, Italy, 1997.

[38] Esteban Feuerstein, Alberto Marchetti-Spaccamela, *Dynamic Algorithms for Shortest Paths in Planar Graphs*, Theory Computer Science, 116 (1993), 359-371.

[39] . Hristo N. Dji Djev, Grammati E. Pant Ziou, and Christos D. Zaroliagis, *Improved Algorithms for Dynamic Shortest Paths*, Algorithmica, 1999.

[40] Lars Arge, *External-Memory Algorithms with Applications in Geographic Information Systems*, Algorithmic Foundations of GIS, Springer-Verlag, Lecture Nodes in Computer Science 1340, 1997.

[41] David Hutchinson, Anil Maheshwari, and Norbert Zeh, *An External-Memory Data Structure for Shortest Path Queries*, Proceedings of International Symposium on Algorithms and Computation, 1999.

[42] N. Zeh, *An External-Memory Data Structure for Shortest Path Queries*, Diplomarbeit, Friedrich-Schiller-Universitit Jena, Nov,1998.

[43] *TIGER/Line Files, 1998.* Technical Documentation, US Department of Commerce Economics and Statistics Administration, Bureau of Census.

[44] *Java Platform Standard Edition, v1.3 API Specification*, http://java.sun.com/j2se/1.3/docs/api

[45] M. L. Fredman, and R. E. Tarjan, *Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms*, Journal of the ACM, 34 (1987), pp.596-615.

[46] Andrew V. Goldberg, Craig Silverstein, *Implementation of Dijkstra's Algorithm Based on Multi-Level Buckets*, Technical Report 95-187, NEC Research Institute, Nov. 1995.

[47] *Dictionary of Algorithms, Data Structures and Problems*, http://hissa.nist.gov/dads/terms.html

[48] Edward P. F. Chan, Kevin Chow, *Multi-scale R-Tree*, CS-TR-99-12, Department of Computer Science, University of Waterloo.