Evaluation of Shortest Path Query Algorithm in Spatial Databases

by

Heechul Lim

A thesis
Presented to the University of Waterloo
in fulfillment of
the thesis requirement for
the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2003

© Heechul Lim 2003

I hereby declare that I am the sloe authour 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

Many variations of algorithms for finding the shortest path in a large graph have been introduced recently due to the needs of applications like the Geographic Information System (GIS) or Intelligent Transportation System (ITS). The primary subjects of those algorithms are materialization and hierarchical path views. Some studies focus on the materialization and sacrifice the pre-computational costs and storage costs for faster computation of a query. Other studies focus on the shortest-path algorithm, which has less pre-computation and storage but takes more time to compute the shortest path. The main objective of this thesis is to accelerate the computation time for the shortest-path queries while keeping the degree of materialization as low as possible.

This thesis explores two different categories: 1) the reduction of the I/O-costs for multiple queries, and 2) the reduction of search spaces in a graph. The thesis proposes two simple algorithms to reduce the I/O-costs, especially for multiple queries. To tackle the problem of reducing search spaces, we give two different levels of materializations, namely, the *boundary set distance matrix* and *x-Hop sketch graph*, both of which materialize the shortest-path view of the boundary nodes in a partitioned graph. Our experiments show that a combination of the suggested solutions for 1) and 2) performs better than the original Disk-based SP algorithm [7], on which our work is based, and requires much less storage than *HEPV* [3].

Acknowledgements

I would sincerely like to thank Dr. Edward P.F. Chan, my supervisor, for his support and encouragement during my research. He gave me countless advices and opinions that enabled me to finish this thesis, which felt like a never-ending work.

I also would like to thank to Dr. M. Tamer Özsu and Dr. Frank Wm. Tompa for their valuable comments, which improved my thesis significantly.

I owe an enormous debt of gratitude towards others who have helped make this thesis possible: Ning Zhang for helping me understand his work and revising the draft of this work and Steve Aengels for showing me how to survive at Waterloo.

Great thanks are also due towards the School of Computer Science and the Bell Laboratory for funding my project. I am also grateful for the many interesting discussions facilitated by the members of Database Research Group.

Finally, I thank my parents for their endless love and support during my study and throughout my life. I would like to dedicate this thesis for them.

Contents

1 Introduction

1.1 The Problems of Previous Studies 1

1.2 Terminology 2

2 Study of Related Works

2.1 Dijkstra's SP Algorithm 9

2.2 Hierarchical Encoded Path Views for Path Query Processing 9

2.3 Disk-Based SP Algorithm..11

2.4 Materialization Trade-Offs in Hierarchical Shortest Path Algorithms 15

2.5 Multiple Range Query 16

3 Algorithms for Improving the Disk-based SP Algorithm

3.1 Search Space Pruning Algorithm using Boundary Set Distance Matrix 21

3.1.1 Boundary Set Distance Matrix 24

3.1.2 Pruning Algorithm Description 25

3.1.3 Proof of Correctness 32

3.1.3.1 Correctness of the β-approximation 32

3.1.3.2 Correctness of the α-approximation 34

3.1.3.3 Correctness of the Pruning Algorithm 38

3.2 Search Space Pruning Algorithm using x-Hop Sketch Graph 39

3.2.1 x-Hop Sketch Graph 40

3.2.2 The Pruning Algorithm an Using x-Hop Sketch Graph..............41

3.2.2.1 Making an Augmented x-Hop Sketch Graph 41

3.2.2.1.1 Properties of an Augmented x-Hop Sketch Graph ...45

3.2.2.2 Calculating the β-approximation 50

3.2.2.2.1 Correctness of the β-approximations 53

3.2.2.3 Calculating the α-approximations 55

3.2.2.3.1 Correctness of the α-approximations ……………56
3.2.2.4 Pruning Boundary Sets ……………………………57
3.3 Query Optimization Using Query Graph ……………………………60
3.4 Shortest-Path Algorithm – Batch Disk-based SP Algorithm ……………63
3.5 An Example of the algorithms ……………………………………………65

4 Experiments

4.1 System Environments and Data sets ……………………………………72
4.2 Query Optimization Using the Algorithm Query Graph ……………………75
4.3 Disk-based SP algorithm vs. Batched Disk-based algorithm ……………76
4.4 Performance with Pruning Algorithms ……………………………………77
4.4.1 Disk-Based SP algorithm ……………………………………78
4.4.1.1 The Effect of Fragment Size ……………………………79
4.4.1.2 The Effect of the Cache Size of the Distance Matrix ……80
4.4.2 Pruning Algorithm Using *BSDistMatrix* ……………………81
4.4.3 Pruning Algorithm Using an *x*-Hop Sketch Graph ……………86
4.4.4 Comprehensive Result ……………………………………………90

5 Conclusion and Future Research

5.1 Conclusion ……………………………………………………………94
5.2 Future Works …………………………………………………………95

Bibliography 96

List of Tables

4.1 Test Set Statistics 73

4.2 The Size (MB) of x-Hop Sketch Graphs 74

4.3 Cache Utilization of using *QueryGraph* algorithm 75

4.4 AverageTime per Query for Pruning Using *BSDistMatrix* 82

4.5 Number of Edges in Augmented x-Hop Sketch Graphs and Time to Calculate 88

4.6 Time to Calculate Skeleton Paths for Different Query Type 88

4.7 Comparison of the β-approximation 88

4.8 Comparison of the α-approximation 88

List of Figures

1.1 Partition of a Graph 8
2.1 Graph After Concatenation 11
2.2 2-level *HEPV* 12
2.3 Super Graph of Disk-based SP Algorithm 14
2.4 Example of Range Query 18
3.1 α-graph and β-graph 23
3.2 How to Make Boundary Set Distance Matrix 25
3.3 How to Calculate the β-approximations 28
3.4 Example of Shortest Path 33
3.5 Probing a Boundary Set X 36
3.6 Example of x-Hop graphs 42
3.7 Making an Augmented 3-Hop Sketch Graph 43
3.8 Necessity of Augmentation 46
3.9 Dijkstra's Algorithm with Mixed Values 52
3.10 Difference Between 1-Hop and 2-Hop Sketch Graph 58
3.11 Query Optimization 61
3.12 Accessed Fragments 64
3.13 Query Graph 65
3.14 β-approximations 68
3.15 Pruning a Boundary Set 69
3.16 Finding the Shortest Path with Pruned Graph 70
3.17 Dealing with Multiple Queries 71
4.1 Number of Fragment DB Accesses 77
4.2 Calculation Time for the Different Size of Fragments 79
4.3 Calculation Time According to Different Cache Sizes 80
4.4 Average Number of Boundary Nodes Closed 84
4.5 Average Calculation Time per Query 85
4.6 Average Number of Boundary Nodes Closed 89

4.7 Calculation Time ……………………………………………………………93
4.8 I/O Activity of Distance Matrix …………………………………………93

Chapter 1
Introduction

1.1 The Problems of Previous Studies

The shortest path problem in very large spatial databases has been elegantly solved to perform relatively well under certain constraints, such as memory and storage requirements [7] [8]. The main idea presented in [7] and [8] is the materialization of existing large databases, such as digital maps or large graphs—which sometimes cannot be fit into the main memory or take too much time to load up to the main memory—and then the application of Dijkstra's shortest path algorithm to the materialized data set. The usefulness of the proposed algorithms in [7] and [8] is that the materialized data are small enough to fit into the main memory and to find the shortest paths without a loss of performance. For the experimental algorithm in [7], their algorithm practically works better than Dijkstra's SP algorithm if we take into account the I/O time to load the whole graph into the main memory. The problem with the algorithm, however, is in the case of multiple queries waiting to be processed. Provided that there is a large enough memory to load the whole graph, the I/O time to load it is a one-time cost for the first query. For all subsequent queries, the only cost is the application of Dijkstra's SP algorithm, which is faster than, or as fast as, the new algorithm.

The answers for speeding up the performance of the algorithm in [7] can be found by minimizing I/O accesses to the data, narrowing search spaces in the graph, and so on. This thesis proposes two simple algorithms to reduce the I/O costs, especially for multiple queries. Some similar attempts to minimize I/O costs in a spatial database have

been done in [9], which deals with multiple range queries. For the problem of reducing search spaces, we give two different levels of materializations. Narrowing search spaces has also been tried in [7], even though the result is not promising.

1.2 Terminology

Many of the terms used in this thesis are adopted from [7], and we will repeat definitions of those again briefly. The details and examples of the terms can be found in [7].

Definition 1. Graph

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

Definition 2. Digital Map

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

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 there are three one-to-one functions f_v: $V_s \rightarrow V$, f_e: $E_s \rightarrow E$, f_w: $W_s \rightarrow W$, such that $\forall e_{ij} \in E_s, f_e(e_{ij}) = (f_v(v_i), f_v(v_j)), f_w(w_s(e_{ij})) = w(f_e(e_{ij}))$.

According to 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 connection between the 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 the corresponding edges in the original graph.

Definition 4. Fragment

A *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.

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

- It is a connected component. For undirected graphs, it is a complete graph; i.e., every pair of vertices has a path connecting them.
- There exists an edge connecting the two vertices in a fragment if, and only if, the two corresponding vertices in the original graph are adjacent.

Definition 5. Partition

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

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 $V_f = V_i \cup V_b$. 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 appear in only one fragment.

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. Each boundary set has its own ID, which is unique.

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: $V_s = \{v_b \mid v_b$ is the boundary vertex in $F_i, i \in [1, n]\}$, $E_s = \{(v_i, v_j) \mid \exists F_k, v_i, v_j \in V_k\}$, $W_s = \{w_s(e_{ij}) \mid w_s(e_{ij}) = min(\{SD_k(e_{ij}) \mid k \in [1, n]\})\}$ 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, $SD_k(e_{ij}) = \infty$.

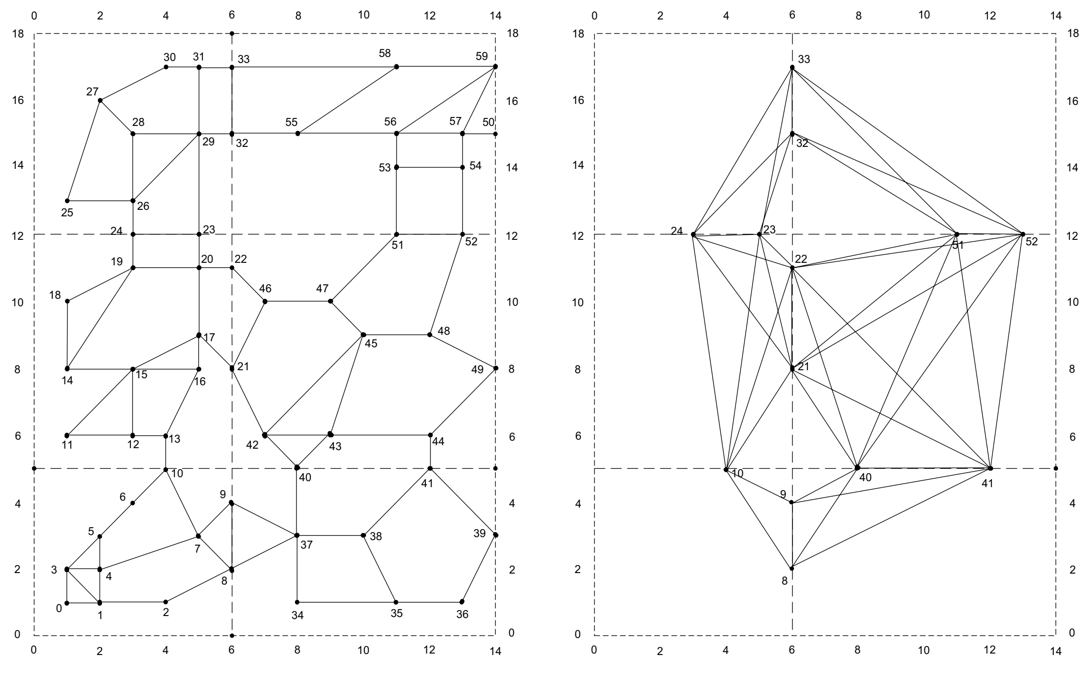
Definition 9. α-value and *minSD*, β-value and *maxSD*

The α-value from 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 a function, $\alpha(S, D) = minSD(S, D) = min(\{SD(v, u) \mid v \in S, u \in D\})$. Similarly, the β-value from a set of vertices S to D can be written as a function, $\beta(S, D) = maxSD(S, D) = max(\{SD(v, u) \mid v \in S, u \in D\})$.

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

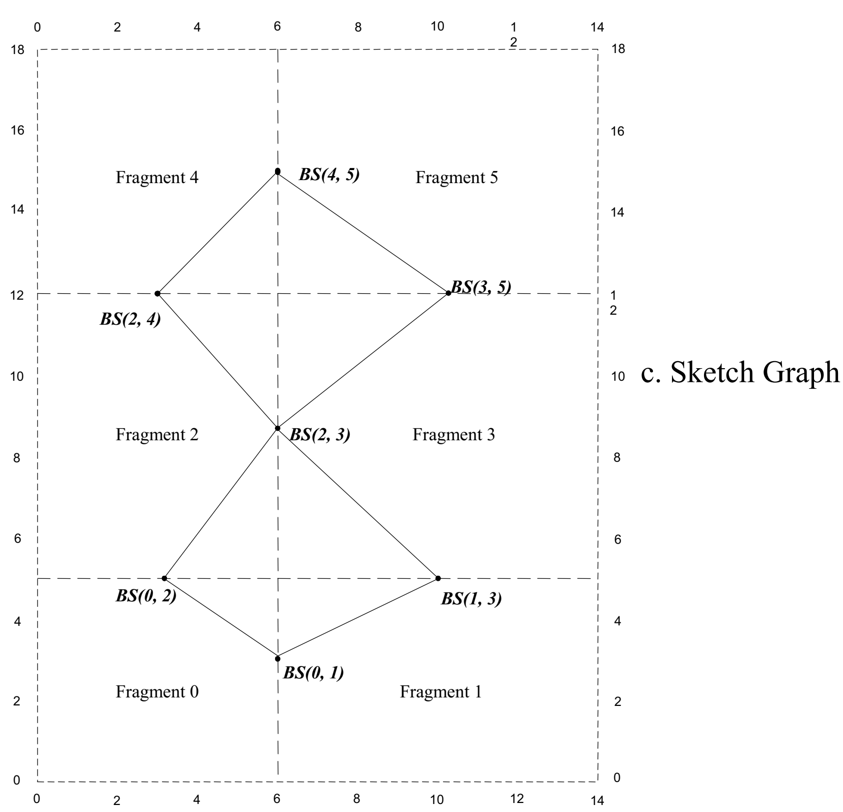
A *sketch graph* $S = (V_s, E_s, W_s)$ of a graph partition $\{F_1, F_2, \ldots , F_n\}$ has the following properties: $V_s = \{v_s \mid v_s$ corresponds to some boundary set in $F_i\}$, that is, there exists a bijection f, where BS_i is the set of boundary sets in the ith fragment F_i. $E_s = \{(v_i, v_j) \mid \exists F_k, f(v_i) \subseteq V_k\}$, 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 a one-to-one function from the set of edges to a set of 2-pair (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 the edges are the α-value of the two boundary sets in super graph, instead of the 2-pair (α, β). Similarly, the β-graph is a sketch graph with the β-values as edge weights.

An example of the partition, including fragments, super graph, and sketch graph, is shown in Figure 1.1.



a. Original Graph and Fragments

b. Super Graph



c. Sketch Graph

Figure 1.1 Partition of a Graph

Definition 11. Boundary Set Distance Matrix (BSDistMatrix)

A *boundary set distance matrix*, or *BSDistMatrix*, is a data structure to contain the shortest path distances between pairs of boundary sets in the sketch graph. The matrix is a square matrix with ($n \times n$) entries, where n is the number of boundary sets in the sketch graph. Each entry in the matrix is a set of 2-pair (α-value, β-value) between two boundary sets.

- $BSDistMatrix = \{(\alpha(v_i, v_j), \beta(v_i, v_j)) \mid v_i, v_j \in V_s\}$, where $\alpha(v_i, v_j)$ and $\beta(v_i, v_j)$ are functions in Definition 9, and V_s is a set of vertices in Definition 10.

Each entry in the matrix gives us a lower and upper bound when we want to calculate an approximation of a shortest distance between two boundary sets. For example, a shortest distance from any vertex in boundary set A to any vertex in boundary set B cannot be less than the α-value of *BSMatrix*[*A*][*B*], and not more than the β-value of *BSMatrix*[*A*][*B*].

Definition 12. α-Approximation, β-Approximation

The α-Approximation from a node u to a node v is defined to be any distance which is equal to, or less than, the actual shortest path distance from u to v. It serves as the lower bound for the actual shortest path distance. The β-Approximation from a node u to a node v is defined to be any distance which is equal to, or more than, the actual shortest path distance from u to v.

Definition 13. Shortest Hop, Shortest Hop Path

The shortest hop refers the shortest distance of the shortest path from a node v_i to a node v_j in a graph, provided that the weights of all the edges in the graph are set to 1. Therefore, if the shortest hop of the shortest path from v_i to v_j in a graph is h, then the path includes h edges, and ($h - 1$) intermediate nodes from v_i to v_j. It can be written as a function, $sh_G(v_i, v_j)$, which returns the shortest hop h from v_i to v_j in a graph G. The shortest hop path naturally means that the path has the shortest hop from the source to the destination. It is certainly possible that more than one shortest hop path exists.

Definition 14. *x*-Hop Sketch Graph, *x*-Hop α-Sketch Graph, *x*-Hop β-Sketch Graph

An *x-Hop sketch graph* $xSG = (V_{xSG}, E_{xSG}, W_{xSG})$ of a sketch graph $S = (V_s, E_s, W_s)$ has the following properties: $V_{xSG} = V_s$. $E_{xSG} = \{e_{ij} \mid e_{ij} = <v_i, v_j>, v_i, v_j \in V_{xSG}, sh_S(v_i, v_j) = x\}$, where $x > 0$, and $sh_S(v_i, v_j)$ is a function which returns the shortest hop h from v_i to v_j in the sketch graph S as in Definition 13. $W_{xSG} = \{w_{xSG} : E_{xSG} \rightarrow (\Re^{\geq 0}, \Re^{\geq 0})\}$, where w_{xSG} is a one-to-one function from the set of edges to a set of 2-pair (α, β), where α are β are the α-value and β-value for the two corresponding boundary sets in the *x*-Hop sketch graph *xSG* respectively.

The sketch graph consists of edges where the two end vertices of each edge are in the same fragment, which means each edge is confined inside the fragment where the two end vertices of the edge are. The *x*-Hop sketch graph of the sketch graph has the same set of the vertices as the sketch graph, but different edges from the sketch graph. The "*x*" in *x*-Hop sketch graph indicates the shortest hops of an edge from a node v_i to a node v_j in the sketch graph. Therefore, the sketch graph *S* is essentially a *1*-Hop sketch graph. In the *2*-Hop sketch graph of *S*, an edge from v_i to v_j denotes the shortest hops of 2 in *S*.

In sum, if the shortest hop from v_i to v_j in the sketch graph *S* is *x*, a new edge from v_i to v_j will be added to the *x*-Hop sketch graph.

An *x*-Hop α-sketch graph is an *x*-Hop sketch graph with edges of α-values. Similar to the α-graph in Definition 10, each edge in the *x*-Hop α-sketch graph represents the minimum shortest distance from any vertex *v* in BS_i to any vertex *u* in BS_j, where BS_i and BS_j are boundary sets that the edge is connecting. Similarly, an *x*-Hop β-sketch graph is an *x*-Hop Sketch graph with edges of β-values.

Definition 15. Augmented *x*-Hop Sketch Graph

The primary use of an *x*-Hop sketch graph is to calculate approximations from a node *s* in *G* to a node *d* in *G*, where *G* is a graph defined in Definition 1. However, the vertices in an *x*-Hop Sketch Graph are not the same as those defined in *G*, because the vertices in an *x*-Hop Sketch Graph represent the boundary sets from a partition of *G*. Therefore, we have to add *s* and *d*, and edges connecting from *s* to the boundary sets in

the fragment F_S where s lies, and from d to the boundary sets in the fragment F_d where d lies, to the x-Hop sketch graph. The weight of an edge from s to BS_i, where BS_i is one of the boundary sets of S, is the minimum shortest distance from s to any boundary node in BS_i. The edges from d to its boundary sets are defined in the same way. Last, after adding s, d, and the edges to their boundary sets respectively, we need to add the edges from the boundary sets of either F_S or F_D to the boundary sets which are h hops away from the boundary sets of F_S or F_D respectively, where $0 < h < x$. An example of the whole process will be given in Section 3.3. Formally, an *augmented x-Hop sketch graph* $axSG = (V_{axSG}, E_{axSG}, W_{axSG})$ of an x-Hop sketch graph $xSG = (V_{xSG}, E_{xSG}, W_{xSG})$ has the following properties: $V_{axSG} = V_{xSG} \cup \{s, d\}$, where s is the source and d the destination. $E_{axSG} = E_{xSG} \cup \{e_{ij} \mid e_{ij} = <s, v_i>, v_i$ corresponds to some boundary set in $S\} \cup \{e_{ij} \mid e_{ij} = <d, v_i>, v_i$ corresponds to some boundary set in $D\} \cup \{e_{ij} \mid e_{ij} = <v_i, v_j>, v_i$ corresponds to some boundary set in F_S, $v_j \in V_{xSG}$, $0 < sh_S(v_i, v_j) < x\}$, where s and d are the source and the destination, and F_S and F_D are the fragments which s and d are in respectively, and $sh_S(v_i, v_j)$ is a function which returns the shortest hop h from v_i to v_j in a sketch graph S as in Definition 13. The definition of E_{axSG} is based on the augmented x-Hop sketch graph which has edges from the boundary sets of F_S to the boundary sets which are h hops away from the boundary sets of F_S. $W_{axSG} = \{w_{axSG} : E_{axSG} \rightarrow (\Re^{\geq 0}, \Re^{\geq 0})\}$, where w_{axSG} is a one-to-one function from the set of edges to a set of 2-tuple (α, β), where α are β are the α-value and β-value for the two corresponding boundary sets in the augmented x-Hop sketch graph $axSG$ respectively.

Chapter 2
Study of Related Works

In Chapter 2, we review the shortest-path algorithms which directly or indirectly inform our work. We first introduce Dijkstra's SP algorithm, the fundamental algorithm for other SP algorithms for extended problems. We then introduce several approaches to solving the problems for various spatial-related queries.

2.1 Dijkstra's SP Algorithm

Dijkstra's SP algorithm is a general method to solve the single-source shortest-path problem [18]. The runtime of the original version was $O(n^2)$, but many studies have been done to improve its performance. One of the best examples is using double buckets on the Dijkstra's algorithm [1]. The new runtime is $O(n\log n)$.

2.2 Hierarchical Encoded Path Views for Path Query Processing

Hierarchical path-finding has been proposed as a solution to the problems of computer networks in [3] and planar graphs in [2]. In [4], a hierarchical routing algorithm called *HEPV*, which offers advantages over alternative path-finding approaches in terms of performance and space efficiency, has been investigated. *HEPV* divides a graph into sub-graphs (fragments), each of which has boundary nodes, and the boundary nodes form a higher-level graph. Edges in the graphs above the ground level are called boundary edges. The cost associated with the boundary edge is the shortest-path cost through the fragment

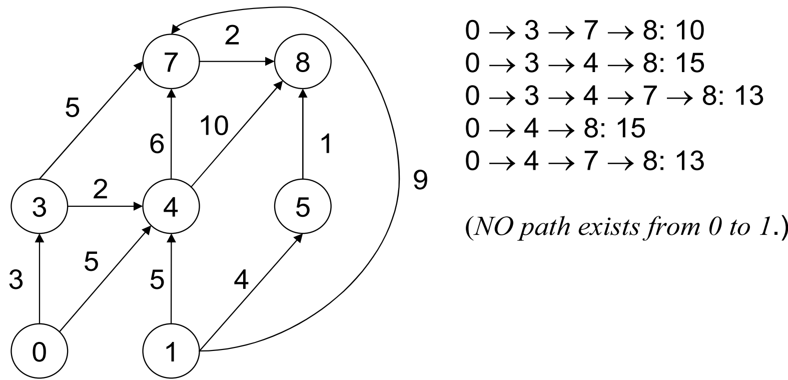
between the boundary nodes. If the size of the higher-level graph is still too large to load into the memory, *HEPV* divides the higher-level graph into another set of fragments. For a large graph, the authours of [4] claim that a three-level *HEPV* is normally efficient in computing shortest-paths.

For each fragment at the ground level, they create and maintain a table called *Encoded Path View*, containing the all-pair shortest-paths. The table stores the origin, destination, direct successor (next hop) node, and the weight for a shortest-path from the origin to the destination. Figure 2.2.(b) shows an example of fragments and their tables. It is straightforward to decode the view from a table. For example, in Figure 2.2, if one wants to find the shortest-path from node 1 to node 3 in fragment 1, one simply looks up the table of node 1 and find the destination, node 3. The row of node 3 in the table indicates the next hop of the shortest-path (node 0) and the weight (4) of the shortest-path. For the graphs above the ground level, we need two more pieces of information to make the tables in addition to the encoded path view of the ground-level graph: 1) the fragment ID of the fragment at the ground level, through which the shortest-path from the origin to the destination first crosses, and 2) the next hop of the shortest-path in that fragment. Figure 2.2.(c) gives an example of a level-1 graph and its encoded path view. For each node in the figure, we have a table, which has encoded path views for all reachable nodes from the node. For example, the shortest-path from node 1 to node 7 in the encoded path view for node 1 passes node 0 as the next hop in fragment 1 where node 0 is, and its weight is 9.

To retrieve the shortest-path from source s to destination d, the algorithm first checks the sub-paths combined by all the boundary node pairs, each of which consists of boundary nodes from fragments to which s and d belong respectively. The algorithm then checks all the paths from s to its boundary nodes in its fragment S, and from the boundary nodes of d's fragment D to d. The shortest-path is simply the concatenation of the shortest-path from s to the boundary node u in S, the shortest-path from u to the boundary node v in D, and the shortest-path from v to d. For instance, in Figure 2.2, we want to find the shortest-path from node 0 in fragment 1 to node 8 in fragment 4. The first step is finding shortest-path pairs from the boundary nodes in fragment 1 to those in fragment 4.

The boundary nodes in fragment 1 are nodes 0, 3, and 4, and the boundary nodes in fragment 8 nodes 4, 5, and 7. There is a total of 6 possible shortest-paths: (1 → 4), (1 → 5), (1 → 7), (3 → 4), (3 → 7), and (4 → 7). Given that, we then concatenate the shortest-paths from the source to the boundary nodes (nodes 3 and 4) in fragment 1 and the shortest-paths from the boundary nodes (nodes 4, 5, 7) in fragment 4 to the destination. The final result of the concatenation would appear as in Figure 2.1. Therefore, retrieving the shortest-path is the same operation as finding the shortest-path from node 0 to node 8 in the graph of the figure. As shown in the example, the scheme of *HEPV* for finding the shortest-path is an exhaustive comparative algorithm, which may cause a problem when the number of levels in *HEPV* becomes large [7].

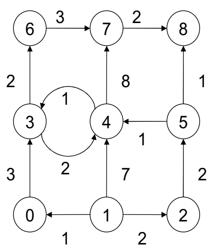
The problem of *HEPV* is that the storage requirement of those views easily reaches more than 2 gigabytes for a relatively small graph of 100,000 nodes with 100 fragments because the *HEPV* approach pre-computes the shortest-paths between all the nodes in each fragment, and the storage requirement may become unacceptable for a larger graph [10] [11].



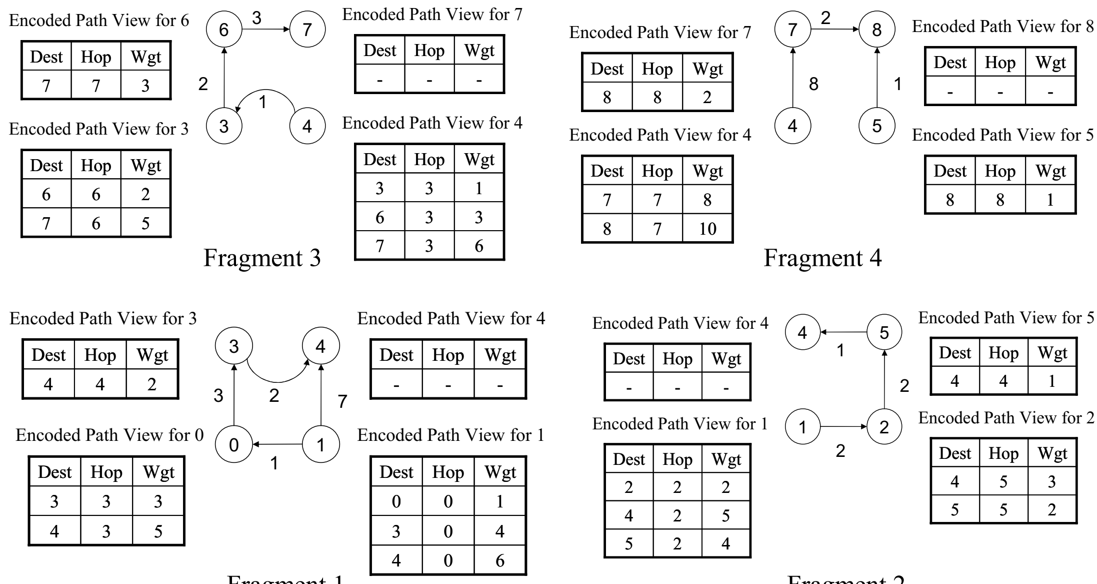
Figire 2.1 Graph After Concatenation

2.3 Disk-based SP algorithm

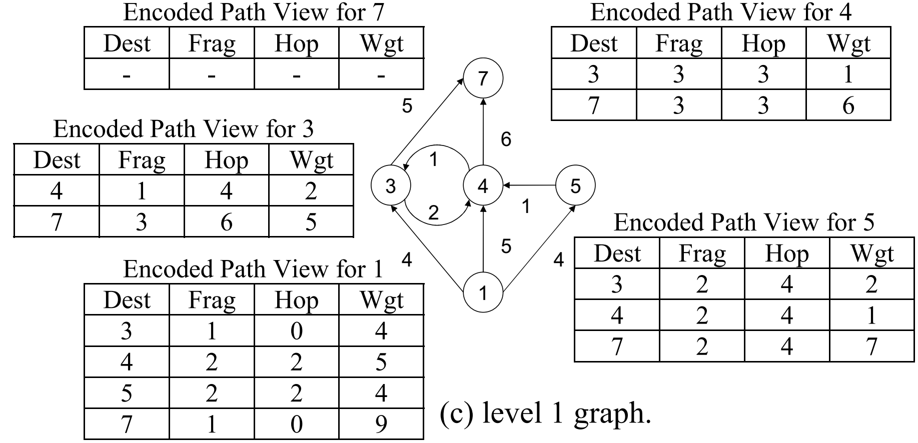
The disk-based SP algorithm in [7] is another variance of Dijkstra's algorithm for a very large spatial database. Similar to *HEPV*, the disk-based SP algorithm divides a graph into fragments, and the boundary nodes form a super graph. The difference from *HEPV* is it uses a different partitioning algorithm, hierarchical scheme, materialization, and shortest-path querying algorithm.



(a) Original Graph.



(b) Fragments and their encoded path views.



(c) level 1 graph.

Figure 2.2 2-level *HEPV*

The algorithm first requires pre-computation for partitioning a graph based on the *BFS* and Hilbert R-Tree [17]. The results of pre-computation are fragments, a super graph, and a sketch graph to capture the outline of the super graph.

For the comparison with *HEPV*, we take the example in Figure 2.2. The original graph is divided into four fragments. Therefore, the fragments are the same as in Figure 2.2.(b) except that the fragments do not have the encoded path views. All-pair shortest-paths are calculated for the boundary nodes of each fragment, and they form the super graph.

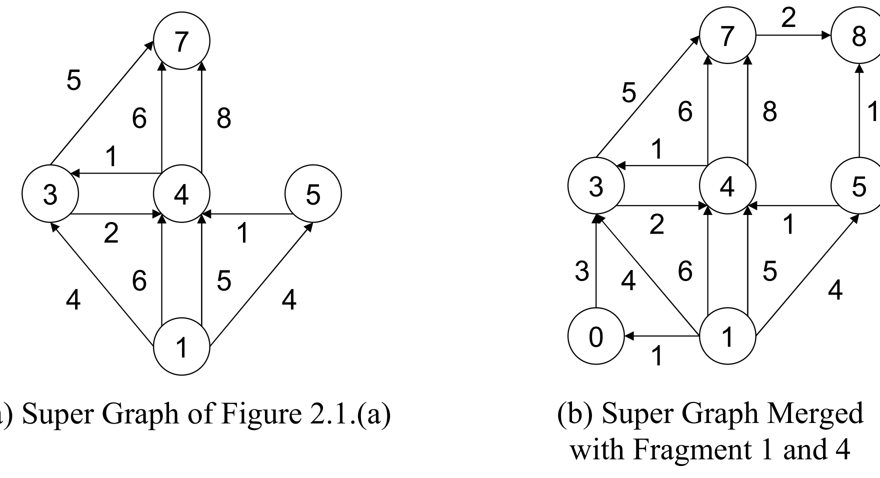
Each edge in the super graph contains the shortest distance between the two end nodes of the edge *within* its fragment. The super graph of the graph 2.2.(a) is shown in Figure 2.3.(a). As a result of the partition, the size of the materialized data, including the fragments and super graph, does not exceed 110% of the size of the original graph, because the fragments and the super graph does not have encoded path view. For a graph of 100,000 nodes, the total storage requirement would be about 15MBytes regardless of the number of fragments, as opposed to more than 2 GBytes in *HEPV*. The query phase of the disk-based SP algorithm largely consists of two parts. The first part is finding a *skeleton path* consisting of boundary vertices only. Intuitively, we calculate the shortest-path by merging the source and destination fragments *S* and *D* with the super graph and then applying Dijkstra's SP algorithm to the merged graph. Figure 2.3.(b) shows a super graph merged with the source fragment 1 and the destination fragment 4. With the merged graph, we apply Dijkstra's SP algorithm to find the skeleton path. The skeleton path of the example will be "$0 \rightarrow 3 \rightarrow 7 \rightarrow 8$."

The second part is finding actual paths. We achieve this by applying Dijkstra's algorithm to the fragments where two consecutive boundary nodes in the skeleton path lie. Since the skeleton path passes nodes 3 and 7 in Fragments 1 and 3 in Figure 2.3, we need to fill out the actual path from 3 to 7 by merging Fragments 1 and 3. We apply Dijkstra's SP algorithm to fill out, and the filled-out path from 3 to 7 is "$3 \rightarrow 6 \rightarrow 7$." In the end, the completed path is "$0 \rightarrow 3 \rightarrow 6 \rightarrow 7 \rightarrow 8$."

The testing result of the disk-based SP algorithm demonstrates that the algorithm needs only less than 60MB of main memory even for a very large graph like the digital

map of East 5 states, whose materialized data comprise around 350MBytes. The average running time for a single query always is always less than that for using the original Dijkstra's algorithm, given that, in Dijkstra's algorithm, the whole graph can be fit into the main memory. If we disregard the I/O time for loading the whole graph, Dijkstra's algorithm is faster than a disk-based algorithm. In practice, a number of queries will come into the system, and loading the graph will be a one-time occasion, which means Dijkstra's algorithm will perform better if we have enough memory to load the whole graph. The main purpose of this thesis, in fact, is to reduce the run-time for the disk-based algorithm while minimizing the costs of pre-computation and additional materialization.

The highest I/O costs are caused by the relaxation process of Dijkstra's algorithm on the super graph during the computation of skeleton paths since the super graph in the disk-based SP algorithm is stored in the external memory. According to [5] [12], the I/O cost of the best-known SP algorithm using external memory was $O(V + (E/B)\log(V/B))$, where V is the number of vertices, E the number of edges, N is the sum of V and E, and B is the number of vertices and edges per disk block. With the disk-based SP algorithm, the I/O cost can be reduced to $O(sort(N))$, where $sort(N) = \Theta(N/B \cdot \log_{(M/B)}(N/B))$, and M is the number of vertices and edges that can be fit into internal memory.



(a) Super Graph of Figure 2.1.(a)

(b) Super Graph Merged with Fragment 1 and 4

Figure 2.3 Super Graph of Disk-based SP Algorithm

2.4 Materialization Trade-offs in Hierarchical Shortest-path Algorithms

A hierarchical shortest-path algorithm decomposes the original graph into a set of fragments and a boundary node graph (super graph) which summarizes the fragment graphs. While a fully materialized hierarchical shortest-path algorithm pre-computes and stores the shortest-path view and the shortest-path-cost view for the fragments as well as for the boundary node graph as we have seen in section 2.2, the storage cost can be reduced by a virtual or hybrid materialization approach, in which few, or none, of the relevant views are pre-computed. The authors in [11] explore the effect of materializing individual views for the storage overhead and the computation time of the hierarchical shortest-path algorithm.

The degree of materialization is divided into two categories in [11].

- Cost View (*CV*)

  The cost (distance) of the shortest-path between all node pairs in the graph. It does not store any path information. For a fragment, a partial materialization of the *CV*, the *C2B* or cost-to-boundary-nodes view stores the cost of the shortest-path from the interior nodes of the fragment to the boundary nodes of the fragment.

- Compressed Path View (*CPV*)

  The set of optimal paths between all nodes on the graph as a series of "hops."

With *CV* and *CPV*, the authors chose four candidates of hybrid materialization for direct comparison in order to facilitate studying the effects of materializing individualization in either the boundary graph or the fragments.

- *F0* has no materialization in either the boundary graph or the fragments.
- *F1* materializes only the *C2B* table in the fragments.
- *F2* materializes the *C2B* table of the fragments and the *CV* table of the boundary graph.
- *F3* materializes the *C2B* table in the fragments and both the *CPV* and *CV* tables in the boundary graph.

Between *F0* and *F1*, they compare the effect of materializing the *C2B* table in the fragments. By comparing *F1* and *F2*, they try to determine the effect of materializing the *CV* in the boundary graph. Comparing *F2* and *F3*, they determine the effect of materializing the *CPV* view in the boundary graph. In sum, *F0* has the least (zero) materialization, followed by *F1* and *F2*. *F4* has the most materialization. For example, the disk-based SP algorithm, discussed in section 2.3, lies between *F0* and *F1* because it materializes only the partial *CV* table of the boundary graph in a fragment. On the other hand, *HEPV*, discussed in 2.2, materializes both the *CPV* and *CV* tables for the boundary graph and the fragments, so it lies beyond *F3*.

They experimented with the CPU-costs, I/O-costs, and storage-costs of the candidate hybrid materialization strategies using the Twin Cities metropolitan road map with 123,000 nodes and 313,000 edges. In terms of the CPU-costs, the number of operations decreases as more views are materialized, which means *F3* performs best. On the contrary, the storage-costs increase with more materialization. In sum, their experimental results show that materializing the shortest-path-cost view (*CV*) for the boundary graph provides the greatest computational savings for a given amount of storage and a small number of fragments, followed by materializing the cost-to-boundary-nodes view for the fragments, and then the shortest-path view for the boundary graph.

2.5 Multiple Range Query

High I/O costs in using large graphs are inevitable given that the system does not have enough memory to load all the necessary data. One of the ways to minimize the I/O-costs is optimizing queries so that queries which may use the same objects to answer will be dealt with together.

The idea of optimizing multiple-range queries in [9] is simple. In order to answer range queries efficiently in 2-D R-trees, the authors in [9] devised various sorting algorithms for those queries. Based on the fact that the processing cost of a range query is affected mainly by the I/O time to fetch the appropriate disk pages, they focus on the I/O activity to manage the queries effectively.

The first approach to servicing a number of requests processes them in a First-Come-First-Served (*FCFS*) manner. In the case of a low rate of query arrivals, *FCFS* is a reasonable service strategy because there is no additional cost to manipulate incoming queries. However, there is a problem with this approach in real-life situations. If the order of processing follows the arrival order, then the probability of having a cache hit will be very low, leading to poor cache utilization. If a number of queries are waiting to be served, we can take a look at them and rearrange them so that the I/O activity is minimal.

Their first attempt to achieve the goal is called Hilbert Sorting (*HS*) [13]. The *HS* algorithm has the following steps:

- For each query, calculate the Hilbert value of the query window's centroid.
- Sort the Hilbert values in increasing order to obtain the total order of the query windows.
- Execute queries in order.

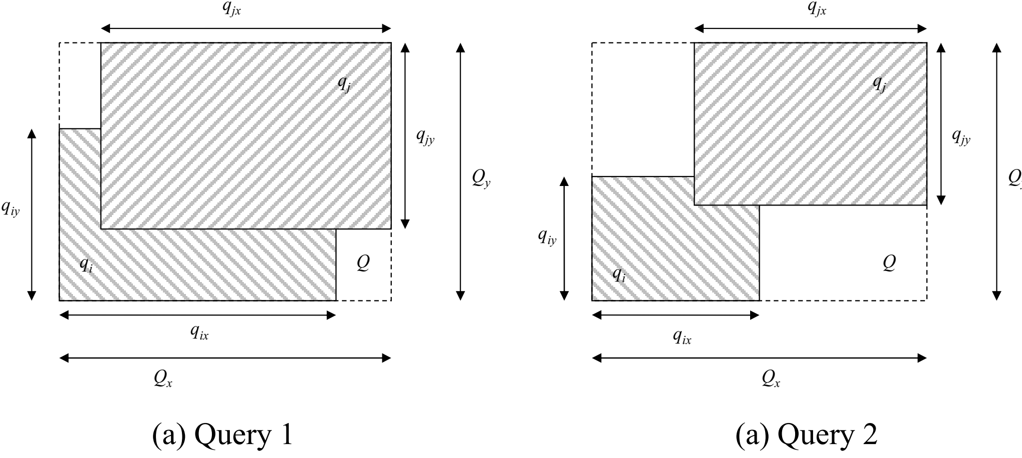
The *HS* method guarantees up to a certain point that nearby requests will be executed sequentially, thus enhancing the locality of references. The pitfall of this method is that it depends heavily on the size of the cache buffer. If there is no buffer space, the algorithm performs the same as the *FCFS* method. For example, assume that two queries, q_1 and q_2, are pending, are next to each other, and are also likely to be sharing a common page. However, if there is no buffer space to store the page, the HS algorithm is useless since the system has to read in the page again for q_2 after q_1 is executed.

To overcome the drawback of the HS method, they first derived an estimate for the expected number of page references for a range query. Let us assume that function $EPR(q_x, q_y)$ returns the expected number of page references for a range query, where q_x and q_y are the x and y extends of the window query q. Intuitively, if the return value of $EPR(q_x, q_y)$ is n, the expected number of page references for the query q is n. Therefore, the smaller n is, the fewer I/O activities are necessary.

First, let us consider two window queries q_i and q_j. If these two queries share common pages, we could execute them as one. What we need is the criterion to decide when to group these queries, or when to execute them individually. They use $EPR(q_x, q_y)$

to determine if the grouping of queries q_i and q_j is advantageous or not. Let Q denote the MBR of the two query windows q_i and q_j. If we execute Q instead of executing q_i and q_j individually, there will be less disk access if and only if $EPR(Q_x, Q_y) \leq EPR(q_{ix}, q_{iy}) + EPR(q_{jx}, q_{jy})$, where x and y for Q, q_i, and q_j are the extends of the queries respectively. The above equation means that the expected number of disk accesses of Q is less than the sum of the ones of q_i and q_j. It is clear that there will be a reduction in the number of disk accesses if the two range queries satisfy the inequality in the above equation. Based on the simple-grouping criterion, they construct two algorithms in which this criterion can be valuable.

Figure 2.4 shows an example of possible scenarios. The dashed-line rectangle represents the Minimum Boundary Rectangle (*MBR*) of the two queries inside. Since the two queries in Figure 2.4.(a) overlap a considerable amount, there is a higher probability that the two queries share more pages. On the other hand, the queries in Figure 2.4.(b) overlap very little, so the two queries are not likely sharing many pages. In terms of the equation $EPR(Q_x, Q_y) \leq EPR(q_{ix}, q_{iy}) + EPR(q_{jx}, q_{jy})$, the example of Figure 2.4.(a) has a greater probability of satisfying the equation. If it satisfies the equation, then grouping two queries and processing them as one query save the I/O activities of reading page references.



(a) Query 1 (a) Query 2

Figure 2.4 Example of Range Query

2.5.1. The Linear Algorithm (Algorithm L)

The idea of this algorithm is that, given two requests, q_1 and q_2, this algorithm will merely check whether the inequality is satisfied or not. If yes, the algorithm will execute the queries as one. If not, it will execute q_1 alone and proceed with q_2 and q_3 until it reaches all pending requests.

Let us assume we have N queries pending.

- For each query, we must calculate the Hilbert values of the window's centroid and then sort the queries according to the Hilbert values in increasing order to obtain the total order of the query windows.
- Let *pos* denote the current query index. Initialize $pos = 1$
- While ($pos < N$) do

    begin
        Test the inequality for query rectangles q_{pos} and q_{pos+1};
        If the inequality is satisfied
        Then process the two queries as one and set $pos = pos + 2$;
        Else process query q_{pos} and set $pos = pos + 1$.
    end
    if *pos* reaches the last query then service q_{pos}.

The complexity of the algorithm is O(Nlog$N$) because of the sorting of the rectangles. After sorting, the algorithm is only O(N) because the queries are scanned only once.

2.5.2. The Extended Linear Algorithm (Algorithm ExL)

The algorithm L considers only two consecutive queries. The authors of [9] extended their idea, enabling the grouping of more than two queries.

Consider the queries $q_1, \dots, q_N$ in increasing order with respect to the Hilbert value of the rectangle centroid. The algorithm tries to pack requests into disjointed sets. The algorithm begins with query q_1. Initially, the first group, $G1$, contains only q_1. If the processing of q_1 and q_2 together retrieves fewer pages than the processing of q_1 plus q_2 under the rule of the inequality, then $G1 = \{q_1, q_2\}$. If the processing of q_3 plus $G1$ retrieves fewer pages than the processing of q_3 plus q_2 plus q_1, then $G1 = \{q_1, q_2, q_3\}$. The algorithm continues with the same process until it reaches a query q_k such that the

expected number of disk accesses $EPR(G1 + q_k) > EPR(G1) + EPR(q_k)$. When this happens, the algorithm sets G1 = $\{q_1, \dots , q_{k-1}\}$ and starts a new group. This process goes until all queries are examined.

- For each query, calculate the Hilbert values of the window's centroid. Sort the queries by the Hilbert values in increasing order to obtain the total order of the query windows.
- Let *pos* denote the current query index. Initialize *pos* = 1. Let GroupId denote the current group. Initialize GroupId = 1.
- While (*pos* < N) do

```
    begin
        Initialize EndOfGroup = False and G_GroupId = {q_pos};
        While (not EndOfGroup) do
            begin
                If (P (G_GroupId + q_pos) < P(G_GroupId) + P(q_pos))
                Then assign q_pos to G_GroupId and set pos = pos + 1;
                Else set EndOfGroup = True and set GroupId = GroupId+1;
                Process as one all q_j's ∈ G_GroupId;
            end
    end
If (pos == N) then service q_pos.
```

Provided that the query windows have already been sorted with respect to the Hilbert values of their centroid, the time complexity of the algorithm is linear to the number of queries O(N), the same as with algorithm L.

The results show that as buffer size increases, the performance of all methods is improved. Also, the more range queries are in pending, the more efficient is the derived processing plan. Another important point is that as the size of the query window increases, so does the performance. Generally, algorithm HS is better in the case of large buffers, while algorithm L is the choice in all other cases. What we learn from [9] is that using simple sorting algorithms can significantly reduce the I/O-costs.

Chapter 3
Algorithms for Improving the Disk-based SP Algorithm

As we have seen in Section 2.3, the disk-based SP algorithm performs slower than the original Dijkstra's SP algorithm once the system has loaded the whole graph. In practice, that scenario is certainly possible if we have enough memory to load the graph. In this chapter, we will present three different approaches to making the disk-based SP algorithm perform better while the additional materialized data necessary to achieve the goal remain as small as possible.

Sections 3.1 and 3.2 discuss algorithms for pruning the search space of a graph. Section 3.1 gives a simple, yet effective pruning algorithm using *BSDistMatrix*. Section 3.2 explains another pruning algorithm using the *x*-Hop sketch graph for pruning search spaces.

Section 3.3 discusses an algorithm for optimizing multiple queries so that the I/O-cost for the disk-based SP algorithm can be minimized.

Section 3.4 explains an algorithm that focuses on minimizing the I/O-cost for finding actual paths.

3.1 Search Space Pruning Algorithm Using Boundary Set Distance Matrix

Dijkstra's shortest path algorithm with proper data structure [12] is effective in finding a path in a graph, and many modifications to the algorithm are made to fit in certain

situations. In [7], the authors developed the disk-based SP algorithm to find the shortest path in a very large network system. The experimental results of their work show that the average running time of their algorithm ranges from about the same as to two-and-a-half times slower than that of Dijkstra's algorithm, provided that, in Dijkstra's algorithm, the whole graph can reside in the system's main memory and is loaded in advance. The authours claimed, however, that if, for each query, the I/O-time for loading the whole graph is counted, their algorithm performs better every time. In the specific case that a whole graph can be fit into the main memory, there are not many advantages to using their algorithm. For the example of the Connecticut area, if multiple-path queries are waiting to be served, and the queries are confined, which can be converted into a digital map of 20 Mbytes and loaded into the main memory, the obvious choice for the specific situation is the traditional Dijkstra's algorithm. Hence, the question is how can we make the disk-based SP algorithm work at least as well as or better than Dijkstra's algorithm.

One of the interesting algorithms in [19] is the graph-pruning algorithm. Even if the proposed algorithm does not improve the performance much, the idea behind it can be easily modified and can improve the performance by materializing additional information during the pre-processing phase. The reason his pruning algorithm in [19] works poorly is that it does not generate good approximations. In his algorithm, the β-approximation between s and d is calculated by Dijkstra's SP algorithm on a β–graph to ensure that the approximation is never *shorter* than the distance of the real shortest path from s to d, which produces the upper bound for the distance of the real shortest path. On the other hand, the α-approximations for pairs from s to all other boundary sets and from all the boundary sets to d are calculated on an α–graph to ensure that each approximation is never *longer* than the distance of the real shortest path for each pair, which produces the lower bound for the distance of the real shortest path from s and d to all other boundary sets. The pruning algorithm is simple. If the approximation based on the α-graph from s to d passing a boundary set X is longer than the approximation based on the β-graph from s to d, then we can remove X safely for the calculation of a skeleton path. Figure 3.1 shows an example of a pitfall of the algorithm. For the convenience of the explanation, we include boundary nodes in each boundary set even if the nodes in the actual sketch

graph represent the boundary sets. Each node in the figure represents a boundary node of a fragment. Each arrow represents a distance between two connected nodes. The figure shows parts of the α- and β-graph of a given graph. If we apply Dijkstra's SP algorithm from the boundary set X to the boundary set Y on the α-graph, the outcome will appear as 3.1 (c). Figure 3.1 (d) shows the shortest path based on the β-graph. The difference between the distances of the two shortest paths is huge because of the definition of the α- and β-values. Since the difference between the α- and β-values even with the same edge of the sketch graph is usually large, the calculation is never expected to give good pruning results.

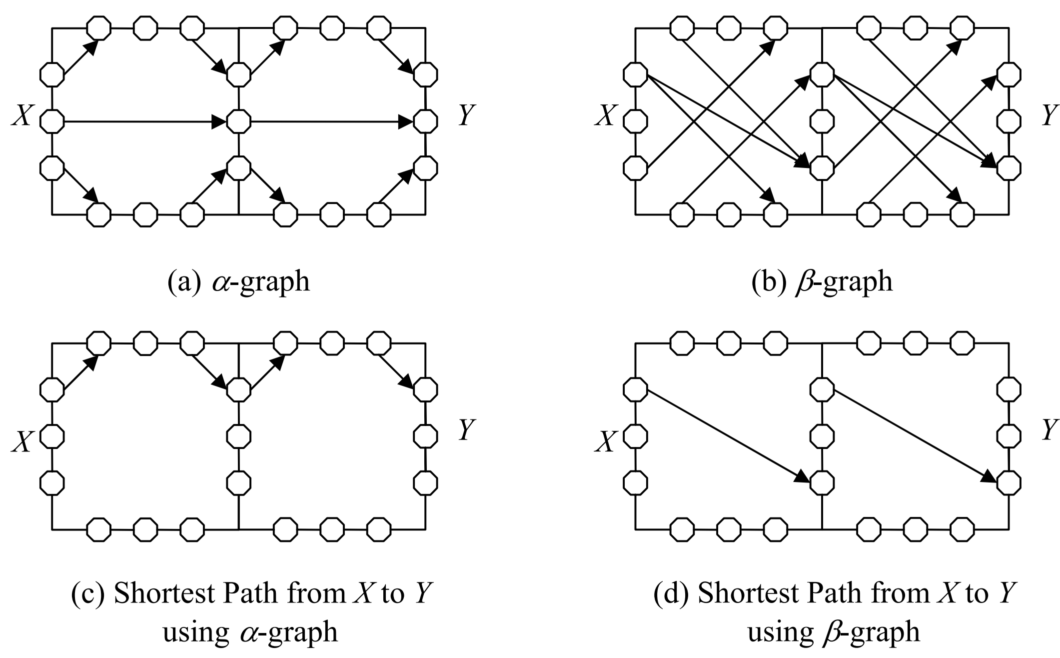


Figure 3.1 α-graph and β-graph

To tackle the problem, we need to use alternative values to obtain good approximations instead of the α- and β-values. The main idea is to calculate all-pair shortest distances for all the boundary node pairs in a super graph. From the calculated distances between the pairs, we can draw the minimum and maximum shortest distances

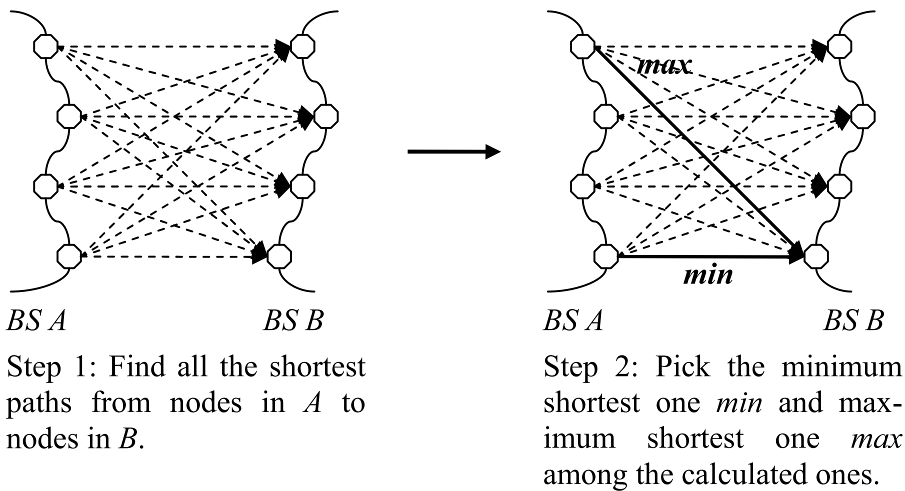
between all the boundary sets, which we will then materialize and store in the secondary storage. Therefore, it would take a long time to generate such data if there were many boundary nodes in a graph to handle. Furthermore, if the boundary sets of the graph are many, the size of the data will be large, and thus the data will be difficult to use. However, properly setting the size will help us generate a reasonable amount of data in a reasonable amount of time.

3.1.1 Boundary Set Distance Matrix

In order to prune a sketch graph efficiently, we need to build a set of matrices storing the shortest distances between the boundary sets in the sketch graph. Since a boundary set has a number of boundary nodes, there exist multiple shortest distances between two boundary sets. Our solution is to keep only the minimum and maximum shortest distances between the two boundary sets. Therefore, the shortest distance from a boundary set A to a boundary set B is defined as the minimum of the shortest distances from any node in A to any node in B. In other words, the minimum shortest distance from a to b is the shortest among the shortest distances from any node in A to any node in B. The maximum shortest distance is defined in a similar manner: it is the longest among the shortest distances from any node in A to any node in B.

To build a *Boundary Set Distance Matrix* (*BSDistMatrix*), we first need to calculate all the shortest distances from the boundary nodes of the first set to the boundary nodes of the other boundary sets. Once all the shortest distances have been calculated, the minimum and maximum ones among them are selected.

Figure 3.2 shows the steps of preparing a *BSDistMatrix* from the boundary set A to the boundary set B. After step 2, the minimum and maximum shortest distance from A to B is set to be *min* and *max* respectively. To build the entire *BSDistMatrix* for A, we follow the same steps for every other boundary set in the graph.



Step 1: Find all the shortest paths from nodes in *A* to nodes in *B*.

Step 2: Pick the minimum shortest one *min* and maximum shortest one *max* among the calculated ones.

Figure 3.2 How to Make Boundary Set Distance Matrix

3.1.2 Pruning Algorithm Description

Our objective in using the pruning algorithm is to eliminate as many boundary nodes as we can so that we can reduce the calculation time of finding skeleton paths. The suggested algorithm basically deals with boundary sets, and if a boundary set is proven to be unnecessary for finding the skeleton path of a query, we can prune the whole boundary set and do not need to include the boundary nodes in that boundary set while finding the skeleton path.

The pruning algorithm is independent of the disk-based SP algorithm because it is applied before the disk-based SP algorithm starts. It first begins with building the shortest-path trees rooted from a source and a destination in their respective fragments. From the shortest-path trees, we know the α- and β-values from the source and the destination to their respective boundary sets in their fragments. After the algorithm prunes some of the boundary sets in the sketch graph, the search for the skeleton path continues with the pruned sketch graph, which does not have unnecessary boundary sets, which could be necessary during the disk-based SP algorithm without pruning.

The pruning process has two parts. First, we estimate the shortest distance (β-approximation) from source to destination and then in order to prune the boundary set,

compare it with the approximations (α-approximations) of a path from the source to the destination passing a specific boundary set. The key to pruning as many boundary sets as possible is to estimate distances well.

For example, let us assume that the shortest distance from a source *s* to a destination *d* is 10. By definition, the β-approximation from *s* to *d* must be equal to, or greater than, 10. If the α-approximation of a path from *s* to *d* passing a boundary set *X* is *more* than the β-approximation, we can prune *X* because the path passing *X* cannot be the shortest path by the definition of α-approximation.

The β-approximation of the shortest distance from the source to the destination consists of the minimum of the two cases:

Case 1 (Figure 3.3 (a))

- the maximum shortest distances from the source node to boundary sets in the source fragment.
- the minimum shortest distances from the boundary sets in the source fragment to the boundary sets in the destination fragment.
- the maximum shortest distances from the boundary sets in the destination fragment to the destination node.

Case 2 (Figure 3.3 (b))

- the minimum shortest distances from the source node to boundary sets in the source fragment.
- the maximum shortest distances from the boundary sets in the source fragment to the boundary sets in the destination fragment.
- the minimum shortest distances from the boundary sets in the destination fragment to the destination node.

We choose the minimum distance of the minimums from the two cases described above. The number of combinations for each case is determined by $2 \times m \times n$, where *m* is the number of the boundary sets in the source fragment and *n* the number of the boundary sets in the destination fragment. By doing that, we guarantee the β-approximation is at least equal to, or more than, the actual shortest distance, so that we can use the

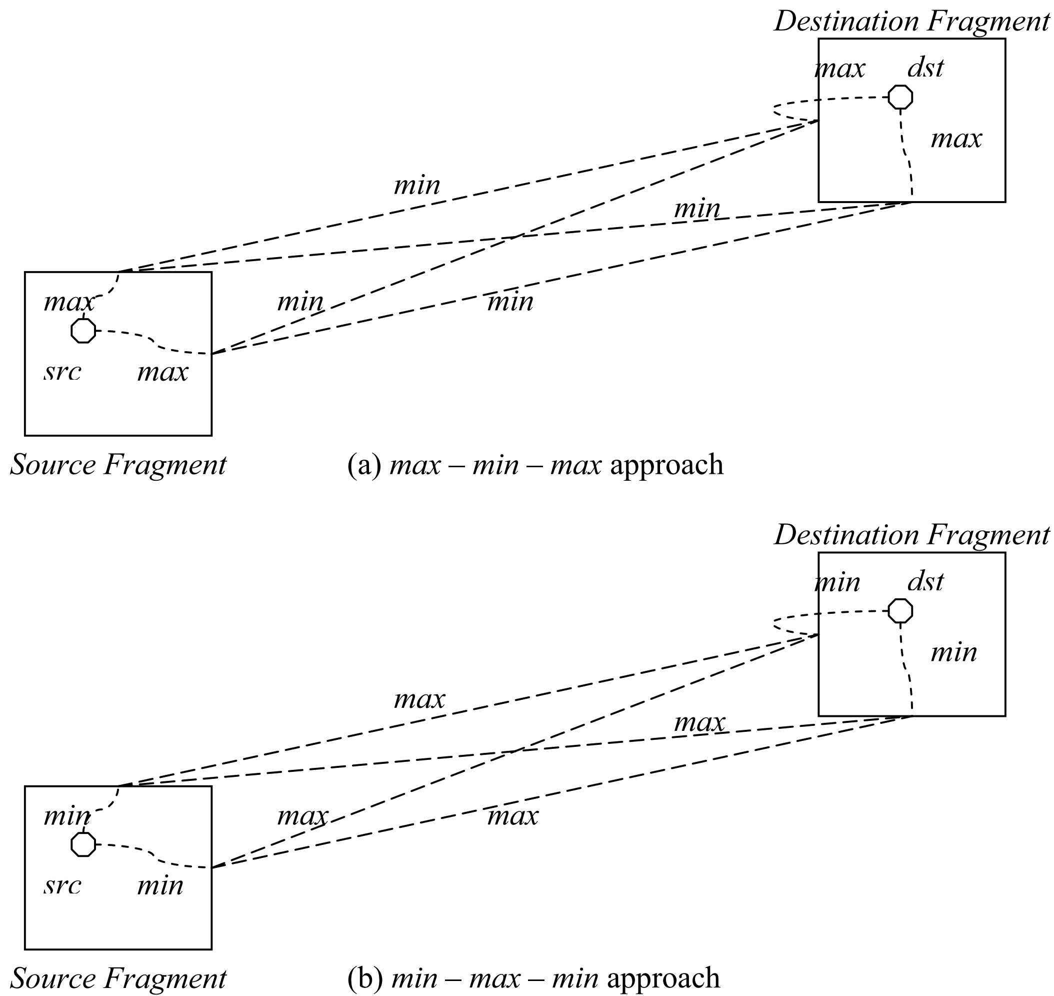
approximation to prune those boundary sets which are parts of the paths having distances of more than the actual shortest distance.

After deciding the β-approximation, we must then calculate an α-approximate shortest distance passing a boundary set to decide whether the boundary set is eligible to be pruned. The process of the α-approximation is similar to the one above. It consists of four parts:

- the minimum shortest distances from the source node to the boundary sets in the source fragment.
- the minimum shortest distances from the boundary sets in the source fragment to a boundary set X chosen to be probed.
- the minimum shortest distances from the boundary set X to the boundary sets in the destination fragment.
- the minimum shortest distances from the boundary sets in the destination fragment to the destination.

We choose the minimum distance of all possible combinations of the four parts above. There are m ways, where m is the number of the boundary sets in the source fragment, to choose from the first two cases, and n ways, where n is the number of the boundary sets in the destination fragment, from the last two cases. Therefore, we have the following number of combinations: $m \times n$. If an α-approximation passing a specific boundary set is longer than the β-approximation, we prune the boundary set.

Since one boundary set usually consists of a number of boundary nodes and each boundary node in a fragment is connected to all other boundary nodes of the fragment in a super graph, pruning one boundary set allows us to eliminate all the boundary nodes in the boundary set and their connected edges to the boundary nodes of all other boundary sets in a fragment.



(a) *max – min – max* approach

(b) *min – max – min* approach

Figure 3.3 How to Calculate the β-approximations

Algorithm 3.1 describes how the disk-based SP algorithm for finding the skeleton path is changed. In fact, no significant changes are made except that the algorithm calls *GraphPrune* algorithm before it starts the main part of the disk-based SP algorithm. Lines 10 to 33 are the main body of the algorithm. When the algorithm finds boundary nodes, it will use *MainThrust*, which is the routine for relaxing all the boundary nodes adjacent to the current close boundary node; otherwise, it relaxes adjacent nodes to the current node,

using super graph. *MainThrust* is the part which takes advantage of the pruning algorithm since *MainThrust* need not relax the boundary nodes of the pruned boundary sets. Lines 6 to 9 are the part preparing and calling the *GraphPrune* algorithm.

Algorithm 3.2 describes how the pruning algorithm is executed. Lines 1 to 11 are the process of calculating the β-approximation from the source to the destination. The process considers every possible combination with given information and chooses the minimum value among them. Therefore, the *approxSP* is the β-approximation to be compared with the α-approximations passing other boundary sets. Lines 12 to 30 are part of the pruning boundary sets. From lines 13 to 19, *sb*, the α-approximation from the source to the currently probed boundary set, is determined, and from 20 to 26, *bd*, the α-approximation from the boundary set to the destination, is determined. The sum of *sb* and *bd* is then compared to the *approxSP*, and whether it has to be pruned or not is described from line 27 to 29.

Algorithm 3.1 FindSkeletonPath (*s*, *d*, *S*, *D*, *M*, *B*, *k*)

Input: *s* and *d* are the source and destination vertices respectively; *S* and *D* are the fragments for *s* and *d* respectively; *M* is the distance matrix database; *BSDM* is the boundary set distance matrix database; and *k* is the sketch graph.

Output: the skeleton path from *s* to *d*.

Precondition: s and d are in *S* and *D* respectively.

/*The distance vector (*dv*) is a data structure of boundary sets to keep track of the shortest distance information from the source. */
1: Initialize distance vector *dv* database
/**bsQ* is an updatable heap. *bsQ* holds the minimum distance information from *s* to each boundary set. *bsQ*.enqueue(*o*) inserts an object *o* into the proper position in *bsQ*. *dv*.delegate() returns objects consisting of all the boundary set IDs and their initial distances (maximum integer) in the sketch graph.*/
2: *bsQ*.enqueue(*dv*.delegate())
3: *s*.distance ← 0
4: *s*.closed ← **TRUE**
/**interQ* is a priority queue, implemented as a binary heap. *interQ* holds the minimum distance information from *s* to nodes in *S* and *D*. *interQ*.enqueue(*node*) inserts a node into *interQ* by its priority of the value of *node*. */
5: *interQ*.enqueue(s)

```
    /*for giving information of min and max distances from the destination*/
6:  build shortest path trees spTreeS, spTreeD for S and D
    /*Get the α and β-values from the source to its boundary sets.
    distS and distD are a simple table containing the minimum and maximum shortest
    distances from the source and destination to their boundary sets
    in their fragments respectively.*/
7:  distS ← spTreeS.getαβValues();
    /*Get the α and β-values from the destination to its boundary sets*/
8:  distD ← spTreeD.getαβValues();
    /*Execute the pruning process.*/
9:  GraphPrune(k, BSDM, S.getBoundarySets(), D.getBoundarySets(), distS, distD);
10: while ~bsQ.isEmpty() ∨ ~interQ.empty() do
        /* interQ.min() and bsQ.min() returns the minimum values of their own.
11:     a ← interQ.min()
12:     b ← bsQ.min()
        /*destination is found*/
13:     if a.equals(d) ∨ b.equals(d) then
14:         break
15:     end if
16:     if interQ.empty() ∨ (b.distance < a.distance) then
          /*relax all boundary vertices adjacent to b*/
17:         do MainThrust on b
18:         b.closed = TRUE
19:     else
            /* interQ.dequeue() removes the minimum-valued object in interQ.
20:         interQ.dequeue()
21:         if a.isBoundaryNode then
22:             find all vertices adjacent to a in S and D and relax them
                /*MainThrust relaxes every boundary node adjacent to a*/
23:             do MainThrust on a
24:         else if a.isInS then
25:             relax all vertices adjacent to a in S
26:         else
27:             relax all vertices adjacent to a in D
28:         end if
29:         a.closed ← TRUE
30:         interQ.enqueue(a)
31:     end if
32: end while
```

Algorithm 3.2 GraphPrune (*k*, *bsM*, *bsInS*, *bsInD*, *distS*, *distD*)

Input: *k* is the sketch graph; *bsM* is the boundary set distance matrix; *bsInS* and *bsInD* are the boundary sets in the source and destination fragments respectively; and *distS* and

distD are the minimum and maximum shortest distance matrices from source to boundary sets in *bsInS* and from destination to boundary sets in *bsInD*, respectively.

Output: Pruned sketch graph.

```
    /*Initialize the β-approximation from source to destination as maximum possible value*/
1:  approxSP ← MAX
    /*start estimating approxSP*/
2:  for all the boundary sets b_S in bsInS do
3:     for all the boundary sets b_D in bsInD do
        /*temp1 is an approximation max – min – max approach (Figure 3.3 (a)). temp2 is
        an approximation min – max – min approach (Figure 3.3 (b)).
        The methods getMax(bs) and getMin(bs) of distS and distD return the maximum
        and minimum values from the source to bs in the source fragment, and
        from the destination to bs in the destination fragment respectively. */
4:        temp1 ← distS.getMax(b_S) + bsM.getMin(b_S, b_D) + distD.getMax(b_D)
5:        temp2 ← distS.getMin(b_S) + bsM.getMax(b_S, b_D) + distD.getMin(b_D)
        /*Choose the minimum between temp1 and temp2.*/
6:        temp = Min(temp1, temp2);
7:        if approxSP > temp then
8:           approxSP = temp
9:        end if
10:    end for
11: end for
    /*probe all the boundary sets in the sketch graph*/
12: for all the boundary sets b in k do
      /*initialize temporary approximation from source to b*/
13:    sb ← MAX
14:    for all the boundary sets b_S in bsInS do
15:       temp ← distS.getMin(b_S) + bsM.getMin(b_S, b)
16:       if sb > temp then
17:          sb ← temp
18:       end if
19:    end for
       /*initialize temporary approximation from b to destination*/
20:    bd ← MAX
21:    for all the boundary sets b_D in bsInD do
22:       temp ← bsM.getMin(b_D, b) + distD.getMin(b_D)
23:       if bd > temp then
24:          bd ← temp
25:       end if
26:    end for
      /*the sum of sb and bd is greater than approxSP*/
27:    if approxSP < (sb + bd) then
```

```
        /*Remove boundary set b from sketch graph k.*/
28:        k.remove(b)
29:     end if
30: end for
```

3.1.3 Proof of Correctness

To prove the correctness of the pruning algorithm, we need to justify three statements:

1. The β-approximation from the source to the destination is equal to, or more than, the actual shortest distance. In other words, the approximation is an upper bound for the actual shortest distance.
2. The minimum distance among the α-approximations from the source to the destination passing nodes in a boundary set X is less than the actual shortest distance passing any node in X.
3. If the minimum approximation of a path passing a node in X is more than the β-approximation from the source to the destination, the actual shortest path cannot pass through any node in X.

Before proving those three statements, let us assume that we have the shortest-path trees for the fragments where the source and the destination belong, and the boundary set distance matrix, which has minimum shortest distances between all boundary set pairs.

3.1.3.1 Correctness of the β-approximation

To prove the first statement, we assume that there is a shortest path between source *src* and destination *dst*, so the shortest distance from *src* to *dst* is *minSD*(*src*, *dst*) according to definition 9 in Chapter 1. The shortest path passes *src* as the starting node, *bn_s* as the first-node boundary node to be passed in the source fragment, *bn_d* as the last-node boundary node to be passed in the destination fragment, and *dst* as the destination node (Figure 3.4).

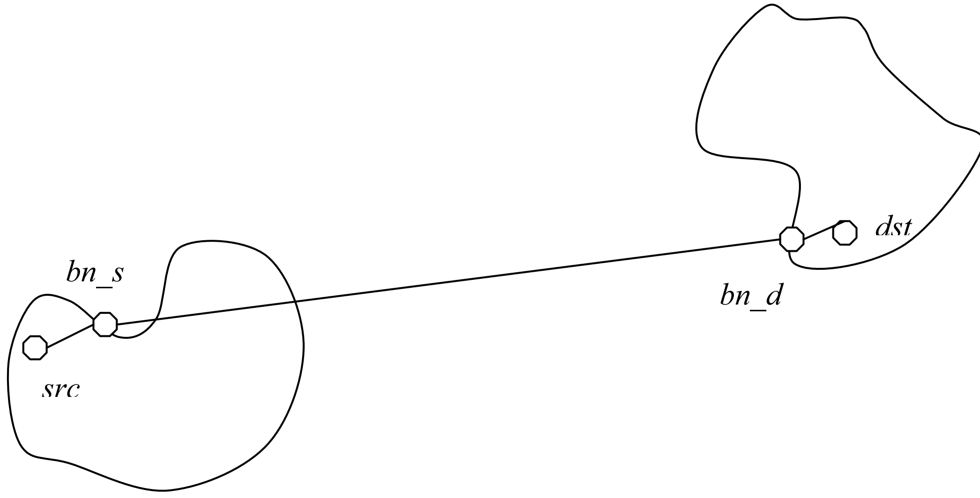


Figure 3.4 Example of Shortest Path

We define the β-approximation for the shortest distance by calculating a minimum of the two cases. If the following two cases satisfy the condition of the β-approximation, then we choose the minimum of the two for the better approximation.

Case 1: $maxSD(src, srcBS_i) + minSD(srcBS_i, dstBS_j) + maxSD(dstBS_j, dst)$, where $srcBS_i$ and $dstBS_j$ are the boundary sets of the source and the destination fragment respectively.

Proof

We assume that the nodes used in $minSD(srcBS_i, dstBS_j)$ are *bn_s'* and *bn_d'*. In other words, the path from *bn_s'* to *bn_d'* gives the minimum shortest distance from $srcBS_i$ to $dstBS_j$. We are now able to derive the following equation.

$minSD(src, bn_s') + minSD(bn_s', bn_d') + minSD(bn_d', dst) \geq minSD(src, dst)$.

Since $maxSD(src, srcBS_i) \geq minSD(src, bn_s')$ and $maxSD(dstBS_j, dst) \geq minSD(bn_d', dst)$, we have

$maxSD(src, srcBS_i) + minSD(srcBS_i, dstBS_j) + maxSD(dstBS_j, dst) \geq minSD(src, dst)$.□

Case 2: *minSD*(*src*, $srcBS_i$) + *maxSD*($srcBS_i$, $dstBS_j$) + *minSD*($dstBS_j$, *dst*), where $srcBS_i$ and $dstBS_j$ are the boundary sets of the source and the destination fragment respectively.

Proof

We assume that the nodes used in *minSD*(*src*, $srcBS_i$) and *minSD*($dstBS_j$, *dst*) are *bn_s'* and *bn_d'*. In other words, the path from *src* to *bn_s'* gives the minimum shortest distance from *src* to $srcBS_j$ and similarly, the path from *bn_d'* to *dst* gives the minimum shortest distance from $dstBS_j$ to *dst*. We are now able to derive the following equation.

minSD(*src*, *bn_s'*) + *minSD*(*bn_s'*, *bn_d'*) + *minSD*(*bn_d'*, *dst*) ≥ *minSD*(*src*, *dst*).

Since *minSD*(*src*, $srcBS_i$) = *minSD*(*src*, *bn_s'*), *minSD*($dstBS_j$, *dst*) = *minSD*(*bn_d'*, *dst*), and *maxSD*($srcBS_i$, $dstBS_j$) ≥ *minSD*(*bn_s'*, *bn_d'*) we have *maxSD*(*src*, $srcBS_i$) + *minSD*($srcBS_i$, $dstBS_j$) + *maxSD*($dstBS_j$, *dst*) ≥ *minSD*(*src*, *dst*). □

Finally, we derive the following *lemma* for the β-approximation.

Lemma 3.1: The β-approximation from *src* to *dst* is equal to, or more than, the actual shortest path from *src* to *dst*.

3.1.3.2 Correctness of the α-approximation

To prove the second statement, we need to consider special cases, such as probing boundary sets in the source and the destination fragments. We categorize the problems in three ways.

Case 1: The minimum distance among the α-approximations from the source to the destination nodes in a boundary set *X* is less than the actual shortest distance passing any node in *X*, where *X* is neither in the source nor in the destination fragments.

Case 2: The minimum distance among the α-approximations from the source to the destination nodes in a boundary set *X* is less than the actual shortest distance passing any node in *X*, where *X* is in the source fragment.

Case 3: The minimum distance among the α-approximations from the source to the destination nodes in a boundary set *X* is less than the actual shortest distance passing any node in *X*, where *X* is in the destination fragment.

To prove Case 1, we assume that there is a shortest path between the source and the destination passing a boundary set *X*, so the path passes *src* as starting node, *bn_s* as the first boundary node to be passed in the source fragment, *x_f* as the first boundary node to be passed in *X*, *x_l* as the last boundary node to be passed in *X*, *bn_d* as the last boundary node to be passed in the destination fragment, and *dst* as the destination node (Figure 3.5 (a)). Therefore, the distance of the shortest path can be calculated by *minSD*(*src*, *bn_s*) + *minSD*(*bn_s*, *x_f*) + *minSD*(*x_f*, *x_l*) + *minSD*(*x_l*, *bn_d*) + *minSD*(*bn_d*, *dst*).

We defined the α-approximation of a path passing *X* by calculating the minimum of the combinations $minSD(src, BS_i) + minSD(BS_i, X) + minSD(X, BS_j) + minSD(BS_j, dst)$, where BS_i and BS_j are the boundary sets in the source and the destination fragments respectively, and *i* and *j* are the number of boundary sets in both fragments. We will prove that the result of $minSD(src, BS_a) + minSD(BS_a, X) + minSD(X, BS_b) + minSD(BS_b, dst)$, where BS_a and BS_b are the boundary sets containing *bn_s*, *bn_d* respectively, is less than the actual shortest path.

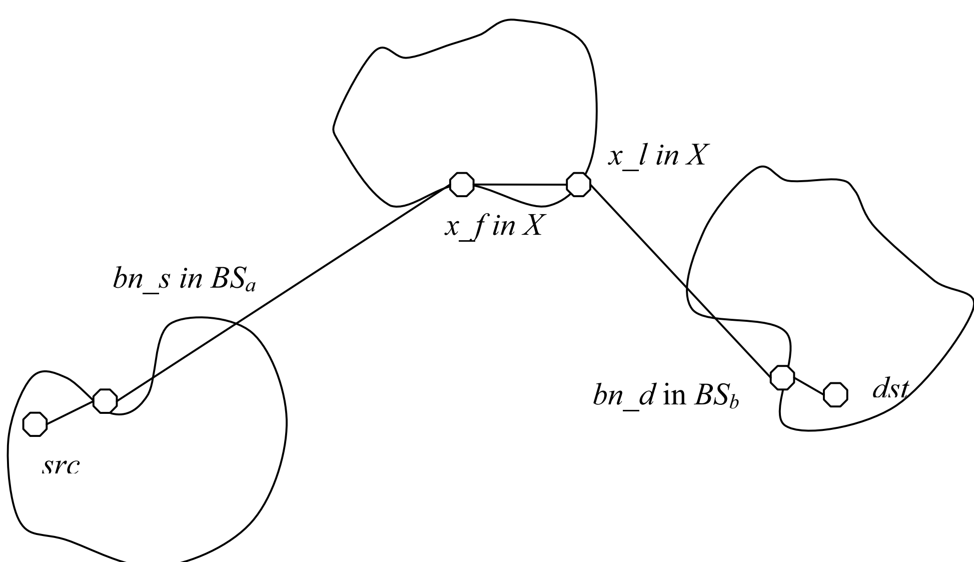
The shortest distance passing *X* is

(iv) *minSD*(*src*, *bn_s*) + *minSD*(*bn_s*, *x_f*) + *minSD*(*x_f*, *x_l*) + *minSD*(*x_l*, *bn_d*) + *minSD*(*bn_d*, *dst*).

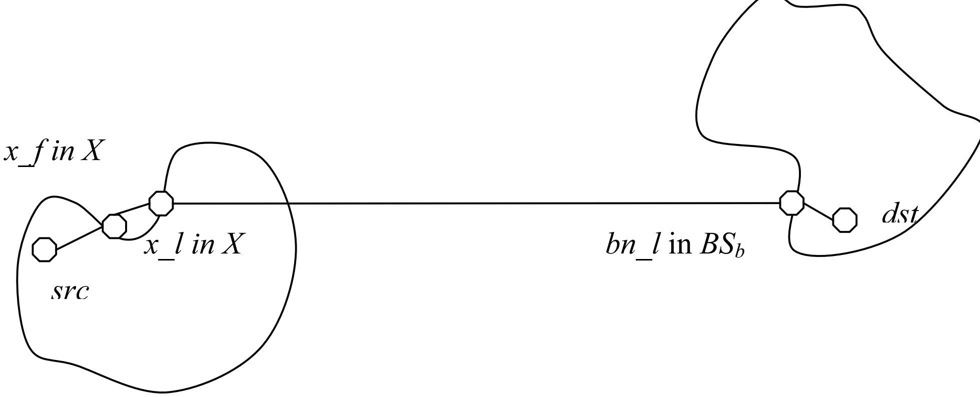
The α-approximation of the path passing BS_a, *X*, and BS_b where BS_a has *bn_s* and BS_b has *bn_d* is

(v) $minSD(src, BS_a) + minSD(BS_a, X) + minSD(X, BS_b) + minSD(BS_b, dst)$.

minSD(*src*, *bn_s*) in (iv) is equal to, or more than *minSD*(*src*, BS_a) in (v) because BS_a contains *bn_s* and *minSD*(*src*, BS_a) calculates the minimum shortest distance from *src* to any node in BS_a including *bn_s*. *minSD*(*bn_d*, *dst*) in (iv) is equal to, or more than, *minSD*(BS_b, *dst*) for the same reason.



(a) *X* is neither in the source fragment nor the destination fragment



(b) *X* is in the source fragment

Figure 3.5 Probing a Boundary Set *X*

minSD(*bn_s*, *x_f*) in (iv) is equal to, or more than *minSD*(BS_a, *X*) in (v) because BS_a contains *bn_s* and *X* does *x_f*, and *minSD*(BS_a, *X*) returns the minimum shortest distance from any node including *bn_s* in BS_a to any node including *x_f* in *X*. *minSD*(*x_l*, *bn_d*) in (iv) is equal to, or more than *minSD*(*X*, BS_b) in (v).

From the above, we have the following equations:

minSD(*src*, *bn_s*) in (iv) ≥ *minSD*(*src*, BS_a) in (v),

minSD(*bn_s*, *x_f*) in (iv) ≥ *minSD*(BS_a, *X*) in (v),

minSD(*x_l*, *bn_d*) in (iv) ≥ *minSD*(*X*, BS_b) in (v),

minSD(*bn_d*, *dst*) in (iv) ≥ *minSD*(BS_b, *dst*) in (v).

In addition, (iv) has *minSD*(*x_f*, *x_l*) added, so finally we get the result of (iv) ≥ (v), which means the α-approximation passing *X* is equal to, or less than the actual shortest distance of the path passing any node in *X*.

To prove Case 2, we assume that the shortest distance of the path is obtained by calculating *minSD*(*src*, *bn_f*) + *minSD*(*bn_f*, *x_f*) + *minSD*(*x_f*, *x_l*) + *minSD*(*x_f*, *bn_l*) + *minSD*(*bn_l*, *dst*) where *bn_f*, *x_f*, and *x_l* are in the boundary sets of the source fragment. Proving that is exactly the same as Case 1. The special case is that *bn_f* is equal to *x_f* (Figure 3.5 (b)). The shortest distance for the special case is *minSD*(*src*, *x_f*) + *minSD*(*x_f*, *x_l*) + *minSD*(*x_l*, *bn_l*) + *minSD*(*bn_l*, *dst*). We already know *minSD*(*src*, *X*), *minSD*(BS_b, *dst*), and *minSD*(*X*, BS_b) are less than *minSD*(*src*, *x_f*), *minSD*(*bn_l*, *dst*), and *minSD*(*x_l*, *bn_l*), respectively, from the proof in Case 1. In addition, *minSD*(*X*, *X*) is "*0*". Therefore, we have the following result.

minSD(*src*, *X*) + *minSD*(*X*, *X*) + *minSD*(*X*, BS_b) + *minSD*(BS_b, *dst*) =

minSD(*src*, *X*) + 0 + *minSD*(*X*, BS_b) + *minSD*(BS_b, *dst*) ≤

minSD(*src*, *x_f*) + *minSD*(*x_l*, *bn_l*) + *minSD*(*bn_l*, *dst*).

Therefore, the α-approximation passing *X*, where *X* is the boundary set of the source fragment can never be more than the actual shortest path passing *X*.

Case 3 is very similar to the above. Finally, we have the following *lemma* for the α-approximation.

Lemma 3.2: The α-approximation of the path from *src* to *dst* passing boundary set *X* is equal to, or less than the actual shortest distance from *src* to *dst* passing *X*.

3.1.3.3 Correctness of the Pruning Algorithm

In Section 3.1.3.1 and Section 3.1.3.2, we proved the first and second statements introduced in the beginning of 3.1.3, which are as follows:

- The β-approximation from the source to the destination is equal to, or more than, the actual shortest distance. In other words, the β-approximation is an upper bound for the actual shortest distance.
- The minimum distance among the α-approximations from the source to the destination passing nodes in a boundary set *X* is less than the actual shortest distance passing any node in *X*.

The third statement can be easily proven using the first and second statements. Let us assume the distance of the shortest path *sd* and the distance of the shortest path passing a boundary set *X sdX*.

If the path passes *X* and the minimum of the α-approximations is *more* than the β-approximation from the source to the destination, then *sdX* is more than the α-approximation by *Lemma* 3.2, and the β-approximation is more than *sd* by *Lemma* 3.1, which means the actual distance of the path passing any node in *X*, is always more than the actual shortest path. Therefore, we can prune *X*.

For the correctness of the pruning algorithm, we prove the following statement.

The boundary sets through which the shortest path passes will not be pruned.

Proof

Let us assume that the shortest path passes *src*, *bn_s*, *x_s*, *x_l*, *bn_d*, and *dst* in order, where *src* is the source, *bn_s* the last node to be passed in the boundary set of the source fragment, *x_s* the first node to be passed in boundary set *X* which is one of the boundary sets the shortest path passes, *x_l* the last node to be passed in *X* and *bn_d* the last node to be passed in the boundary set of the destination

fragment, and finally *dst* the destination. Then, the distance *sp* of the path can be calculated as follows:

(i) *minSD*(*src*, *bn_s*) + *minSD*(*bn_s*, *x_s*) + *minSD*(*x_s*, *x_l*) + *minSD*(*x_l*, *bn_d*) + *minSD*(*bn_d*, *dst*) = *minSD*(*src*, *dst*).

If we show the α-approximation passing *X* is less than the actual distance, then we cannot prune *X* since it does not satisfy the third statement. Consider the α-approximation calculated by the following.

(ii) *minSD*(*src*, BS_a) + *minSD*(BS_a, *X*) + *minSD*(*X*, *X*) + *minSD*(*X*, BS_b) + *minSD*(BS_b, *dst*), where BS_a contains *bn_s*, and BS_b contains *bn_d*.

(ii) can be rewritten as follows:

(iii) *minSD*(*src*, BS_a) + *minSD*(BS_a, *X*) + *minSD*(*X*, BS_b) + *minSD*(BS_b, *dst*) since *minSD*(*X*, *X*) = 0.

minSD(*src*, *bn_s*) in (i) ≥ *minSD*(*src*, BS_a) in (iii) by the definition of *minSD*,

minSD(*bn_s*, *x_s*) in (i) ≥ *minSD*(BS_a, *X*) in (iii) by the definition of *minSD*,

minSD(*x_l*, *bn_d*) in (i) ≥ *minSD*(*X*, BS_b) in (iii) by the definition of *minSD*,

minSD(*bn_d*, *dst*) in (i) ≥ *minSD*(BS_b, *dst*) in (iii) by the definition of *minSD*.

Lastly, *minSD*(*x_s*, *x_l*) in (i) = 0 if *x_s* = *x_l*; otherwise, *minSD*(*x_s*, *x_l*) > 0.

As shown in the comparison, (i) cannot be less than (iii), which means the α-approximation passing *X* is *never more* than the actual shortest distance; therefore, *X* cannot be pruned if *X* is a boundary set through which the shortest path passes.□

3.2 Search Space Pruning Algorithm Using *x*-Hop Sketch Graph

The pruning algorithm presented in Section 3.1 takes advantage of the pre-computed shortest distance information for all-pair boundary sets. As the test results show in Chapter 4, the algorithm works very well, eliminating up to 70% of boundary sets out of the total boundary sets. There are, however, two problems when one applies the algorithm.

The first problem is the calculation time for *BSDistMatrix*. To build one entry in a *BSDistMatrix*, we have to calculate all possible pairs of boundary nodes from two

boundary sets which the entry represents. If the graph is huge and each boundary set has a relatively large number of boundary nodes, the calculation time will grow significantly. For example, if each boundary set holds an average of n boundary nodes and the total number of boundary sets is t, then we need to calculate the shortest path tree $(n \times n)$ times to fill out one entry of the *BSDistMatrix*, $(n \times n \times t)$ times to fill out an entire row of the *BSDistMatrix*, and $(n \times n \times t \times (t - 1))$ to fill out all the entries. The other problem is storage space. Since the *BSDistMatrix* stores every possible pair from the entire boundary sets, there should be enough space to store $O(n^2t^2)$ entries, where t is the number of boundary sets in a partitioned graph. For the *BSDistMatrix* to be effective, t should be large, which in turn requires a large amount storage space. On the other hand, if we make the number of boundary sets fewer to save storage space, the *BSDistMatrix* with fewer boundary sets will not be as. Final problem is updating the graph. Even with a small change in the graph, we must build a whole *BSDistMatrix* again because we do not have information about which part has been affected by the change.

An x-Hop sketch graph, defined in Chapter 1, can be an alternative solution to *BSDistMatrix*. While *BSDistMatrix* has the shortest distance information of all-pair boundary set shortest paths in the sketch graph, the x-Hop sketch graph has distance information from one boundary set to a limited number of boundary sets, limited by the number of hops. Therefore, by controlling x, we can adjust the calculation time and storage space for the materialized data.

Since a node in an x-Hop sketch graph does not each all other nodes, we cannot use it to prune the graph in the same way as we use *BSDistMatrix*. Instead, we need a different scheme to calculate approximations. We will apply Dijkstra's SP algorithm on an x-Hop sketch graph.

3.2.1 *x*-Hop Sketch Graph

As defined in Chapter 1, an x-Hop sketch graph is another form of sketch graph with different edges from the original sketch graph. Each edge in the x-Hop sketch graph connects two nodes, which would be apart from each other by x hops in the original

sketch graph. As in the original sketch graph, each edge in the x-Hop sketch graph has α- and β-values.

An x-Hop sketch graph naturally has more edges than the original sketch graph does since the number of neighbor nodes of a node grows as x grows in an x-Hop sketch graph. However, the size of the materialized data of an x-Hop sketch graph with the proper setting of x is normally less than the one of *BSDistMatrix*. The number of entries of an x-Hop sketch graph is kn, while the one of *BSDistMatrix* is always n^2, where n is the number of nodes (boundary sets) and k is the average number of neighbor nodes of a certain node in an x-Hop sketch graph.

Figure 3.6 depicts an example of x-Hop sketch graphs. Each grid in the figure represents a fragment and the edges of each grid represent boundary sets.

3.2.2 The Pruning Algorithm Using an x-Hop Sketch Graph

The process of the pruning algorithm with an x-Hop sketch graph consists of 4 steps.

- Making an augmented x-Hop sketch graph.
- Calculating the β-approximation from source to destination.
- Building the shortest path tree from the source and the destination on an augmented x-Hop α-sketch graph.
- Pruning nodes in the sketch graph.

3.2.2.1 Making an Augmented x-Hop Sketch Graph

In order to apply Dijkstra's SP algorithm to calculate approximations on an x-Hop sketch graph, we must do the following.

We must add the source and the destination of a query of which we want to find the shortest path. Since an x-Hop sketch graph consists of boundary sets as nodes, we need the source and the destination nodes to calculate the approximations in the graph. After adding those nodes, we add edges which are connecting them to the boundary sets of the fragments which the source and the destination are in respectively. The weights of those edges are the α- and β-values. Last, we add edges to the boundary sets in the source

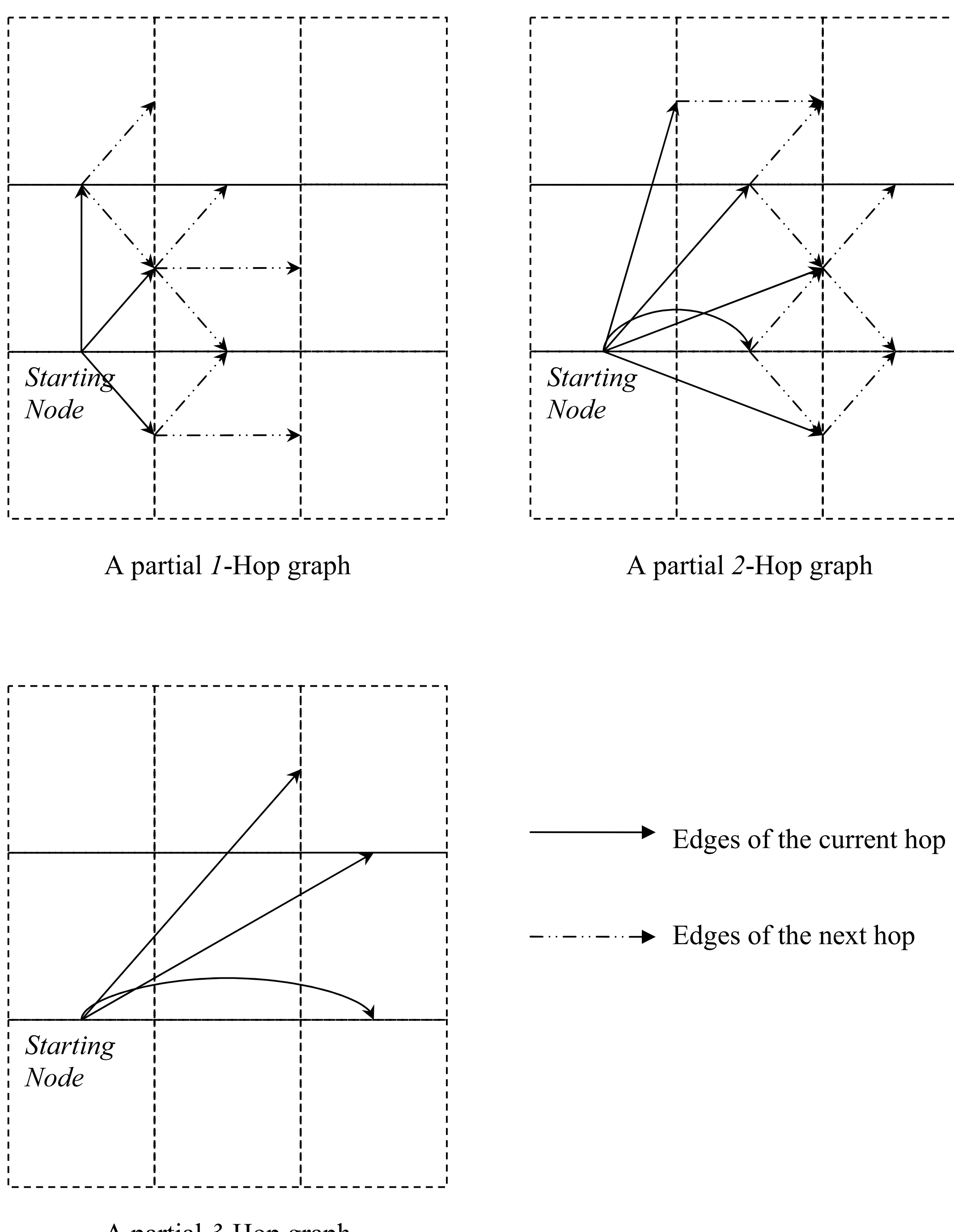
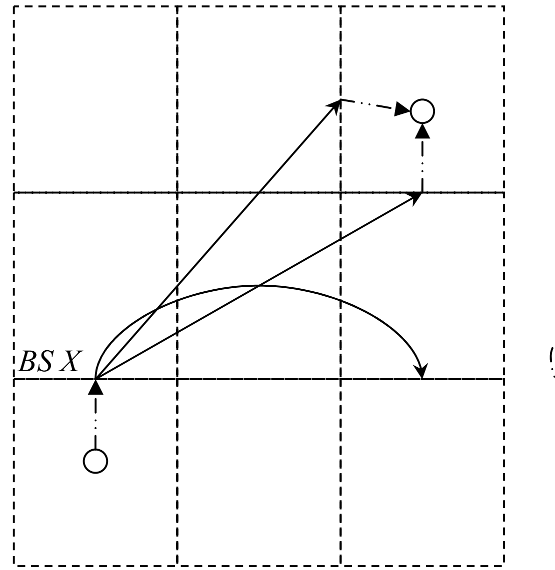
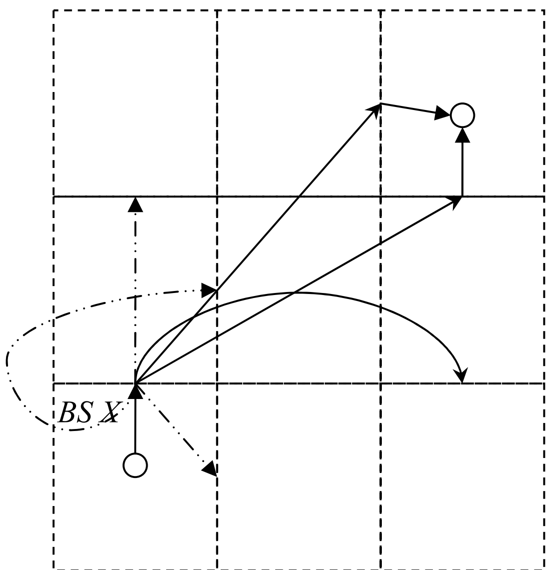


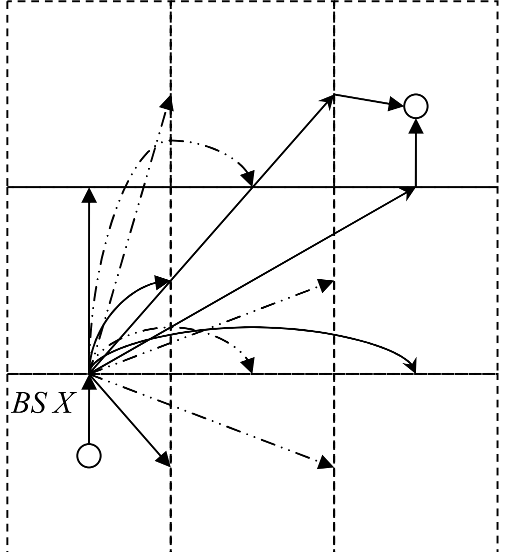
Figure 3.6 Example of *x*-Hop graphs



(a) Adding source and destination, and their edges to *3*-Hop graph



(b) Adding edges from *1*-Hop sketch graph to *3*-Hop graph



(c) Adding edges from *2*-Hop sketch graph to *3*-Hop graph

Already added edges

Currently added edges

The pictures show the process of adding nodes and edges to the *3*-Hop sketch graph.
Figure 3.7 (a) shows adding the source and destination, and edges from the source and the destination to their boundary sets respectively.
Figures 3.7 (b) and (c) show adding edges from *1* and *2*-Hop sketch graphs. It only shows edges from a boundary set *X* in the source fragment, but in practice, we have to add edges of all the boundary sets of the source fragment.

Figure 3.7 Making an Augmented 3-Hop Sketch Graph

fragment, and the edges are from *k*-Hop sketch graphs for all *k*, where $0 < k < x$. The resulting graph (Figure 3.7 (c)) is called an *augmented x-Hop sketch Graph*.

Figure 3.7 shows an example of the augmentation processes for a *3*-Hop sketch graph. For the sake of simplicity, it shows the process of attaching edges from boundary set *X* of the source fragment. As in the example, we first add the source and the destination nodes in the graph and then connect them to the boundary sets in the source and destination fragments. The next steps are simply adding the edges of the boundary sets in the source fragments from the *1* and *2*-Hop sketch graphs.

Algorithm 3.3 shows the pseudo-code for making an augmented *x*-Hop sketch graph. Lines 1 to 18 are the code for adding source and destination nodes, and edges to the boundary sets in the source and destination fragments. Lines 19 to 29 are the code for adding the edges of boundary sets in the source fragment in the *k*-Hop sketch graph, where $0 < k < x$.

Algorithm 3.3 MakeAugmentedXHopSG(*xHopSG, src*, *dst*, *srcF*, *dstF*)

Input: *xHopSG* is an array containing 1… *x*-Hop sketch graph, where *x* is the number of hops, *src* a source node, and *dst* a destination node. *srcF* is a fragment where *src* is, and *dstF* is a fragment where *dst* is.

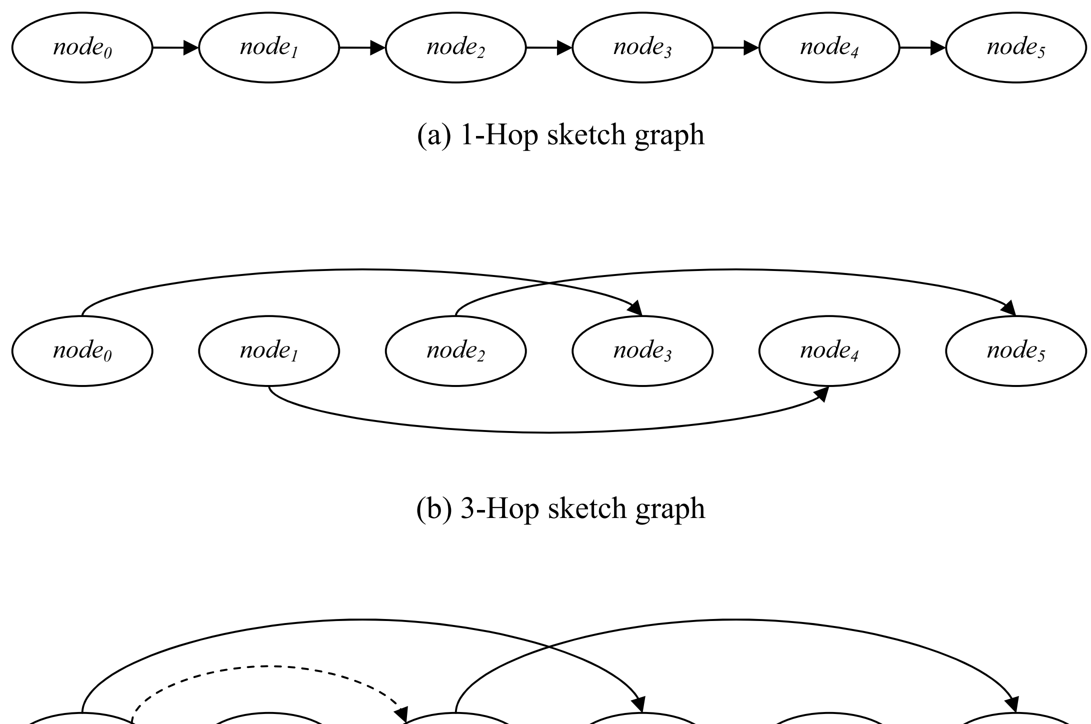
Output: *x*-Hop sketch graph with added nodes and edges

/*Adding src and dst to the x-Hop sketch graph*/
1: *xHopSG*[*xHopSG.length* – 1].*addNode*(*src*)
2: *xHopSG*[*xHopSG.length* – 1].*addNode*(*dst*)
/*Get the shortest path tree rooted from *src* in the source fragment*/
3: SPT_S ← *Dijkstra*(*src*, *srcF*)
/*Get the shortest path tree rooted from *dst* in the destination fragment*/
4: SPT_D ← *Dijkstra*(*dst*, *srcD*)
/*Get the boundary sets in the source fragment*/
5: *bsSetsInSrcF*[] ← *srcF.getBoundarySets*();
/*Get the boundary sets in the destination fragment*/
6: *bsSetsInDstF*[] ← *dstF.getBoundarySets*();
/*Adding edges connecting from *src* to its boundary sets*/
7: **for** all the boundary sets *bs* in *bsSetsINSrcF* **do**
8: *anEdge* ← **new** *Edge*(*src*, *bs*)
/*Set α value for the new edge*/
9: *anEdge.setMin(*SPT_S*.getMin*(*bs*))

```
       /*Set β value for the new edge*/
10:    anEdge.setMax(SPT_S.getMax(bs))
11:    xHopSG[xHopSG.length – 1].addEdge(anEdge)
12: end for
    /*Adding edges connecting from dst to its boundary sets*/
13: for all the boundary sets bs in bsSetsINDstF do
14:    anEdge ← new Edge(bs, dst)
       /*Set α value for the new edge*/
15:    anEdge.setMin(SPT_D.getMin(bs))
       /*Set β value for the new edge*/
16:    anEdge.setMax(SPT_D.getMax(bs))
17:    xHopSG[xHopSG.length – 1].addEdge(anEdge)
18: end for
    /*Adding edges from boundary sets in the source and the destination fragment to
    (i + 1) hop away edges from (i + 1)-Hop sketch graphs*/
19: for elements from i = 0 to i = x – 1 in xHopSG do
       /*Add edges from the source boundary sets*/
20:    for all the boundary sets bs in bsSetsInSrcF do
          /*Get all the adjacent edges of bs*/
21:       edges ← xHopSG[i].getEdges(bs)
          /*Add all the edges to the x-Hop sketch graph*/
22:       xHopSG[xHopSG.length – 1].addEdges(edges)
23:    end for
24: end for /*Have done the sketch graph preparation process*/
```

3.2.2.1.1 Properties of an Augmented *x*-Hop Sketch Graph

The reason for making an augmented *x*-Hop sketch graph is to calculate the approximations which satisfy the conditions of definition 12 in Chapter 1. If we apply Dijkstra's SP algorithm directly on an *x*-Hop sketch graph, we cannot find shortest paths correctly in some cases. Figure 3.8 shows a simple example which underscores the necessity of the augmentation. When we want to find the shortest path from $node_0$ to $node_4$, there is no path between those two nodes in the 3-Hop sketch graph, even if the path does exist in the original sketch graph. If we add an edge of $node_0$ in the 1-Hop sketch graph and the 2-Hop sketch graph to the 3-Hop sketch graph (Figure 3.8 (c)), then we can find the shortest path from $node_0$ to $node_4$. In fact, any path which can be found in the original sketch graph can also be found in the augmented graph.



(a) 1-Hop sketch graph

(b) 3-Hop sketch graph

(c) Augmented 3-Hop sketch graph

Figure 3.8 Necessity of Augmentation

We derive two new *lemmas*, which will be used to prove the correctness of the pruning algorithm using an *x*-Hop sketch graph.

If we convert all the boundary nodes of a skeleton path into the boundary sets in which the boundary nodes are contained respectively, we obtain a "boundary set skeleton path." Since a sketch graph has nodes as boundary sets, each node in a boundary set skeleton path is a boundary set which contains a boundary node in a skeleton path. Let us assume that we have a skeleton path *SKP* of which boundary nodes are bn_0, bn_1, bn_2 …, bn_n. If we have a simple function $BS(bn_i)$ which returns a boundary set ID in which a boundary node bn_i is contained, then we obtain a boundary set skeleton path of *SKP* by

applying the function for every boundary node in *SKP*. The order of nodes in a boundary set skeleton path is the same *total order* as in a skeleton path, which means that, if a node v_i precedes a node v_j in a skeleton path, a boundary set BS_i containing v_i precedes a boundary set BS_j containing v_j in the corresponding boundary set skeleton path. Each edge in a boundary set skeleton path has an α-value and β-value.

Lemma 3.3: We can express any skeleton path of a shortest path of a query as a boundary set skeleton path consisting of nodes and edges of the augmented *1*-Hop sketch graph.

Proof

A skeleton path, by definition, consists of a sequence of nodes and edges from two fragments *S* and *D* in which source and destination are contained respectively, and the super graph. The partial path from the source to the first boundary node in *S*, which the path is passing, can be simplified by an edge from the source to the boundary set in which the first boundary node is contained. In the same manner, we can express the partial path from the last boundary node in *D* to the destination as an edge from the boundary set in which the last boundary node is contained to the destination. The partial path represented by nodes of the super graph is simply a sequence of boundary nodes. Since the sketch graph is a simplified form of the super graph using boundary sets as nodes, the boundary nodes and edges connecting any pair of nodes of the path can be converted to the boundary sets in which they are contained. As a result, we can express all nodes and edges in the shortest path as the nodes and the edges in the augmented sketch graph. □

In a boundary set skeleton path, there is a possibility that some of nodes in the path appear more than once. For example, let us assume that we have a skeleton path $src \rightarrow bn_0 \rightarrow bn_1 \rightarrow bn_2 \rightarrow \ldots \rightarrow dst$, where bn_0 and bn_2 are in a boundary set BS_0, and bn_1 in BS_1. Then, we have a boundary set skeleton path of the skeleton path, $src \rightarrow BS_0 \rightarrow BS_1 \rightarrow BS_0 \rightarrow \ldots \rightarrow dst$, which cannot be generated by Dijkstra's SP algorithm with a sketch graph because the shortest path calculated by Dijkstra's SP algorithm cannot have duplicated nodes in its path. In order to derive further *lemmas*, we need to simplify this

kind of skeleton paths. In the example above, we can simplify the boundary set skeleton path by eliminating one of BS_0, which is closer to *src*, and BS_1, so we have a boundary set skeleton path, *src* → BS_0 … → *dst*. To generalize the idea, we derive *lemma* 3.4.

Lemma 3.4: If we find two or more duplicated nodes, that have the same boundary set ID in a boundary set skeleton path, we can eliminate all the nodes between the first-appearing duplicated node and the last-appearing duplicated node in the path, and then eliminate the first-appearing duplicated node. If we repeat this for all duplicated nodes in the path, we have a *simplified boundary set skeleton path*.

A *simplified* boundary set skeleton path also keeps the same *total order* as the boundary set skeleton path for the nodes existing in both paths, because the simplified boundary set path is derived from the boundary set skeleton path by eliminating some of its nodes.

Lemma 3.5: We can express any skeleton path of a shortest path as a simplified boundary set skeleton path consisting of nodes and edges in an augmented *x*-Hop sketch graph.

Proof

Nodes from the source to the first boundary node in *S* and from the last boundary node in *D* to the destination can be dealt with in the same way as in *Lemma* 3.3. We focus on representing boundary nodes and edges, connecting them in the path using nodes and edges in the augmented *x*-Hop sketch graph. Let us consider a skeleton path *P* of a shortest path and assume that *P* passes *src* → bn_0 → bn_1 → bn_2 → … → *dst* in the super graph. We can convert *P* to a simplified boundary set skeleton path *P’* by *Lemmas* 3.3 and 3.4.

We now consider a new path *P”* in the augmented *x*-Hop sketch graph, where $x = k$. *P”* must have the same *total order* as *P’* for the nodes existing in both *P”* and *P’*.

We give a short algorithm that converts *P’* to *P”* while the same *total order* as in *P’* for *P”* is preserved.

```
// P' is a simplified boundary set skeleton path.
// xSG is an augmented x-Hop sketch graph in which we want to find
// a simplified boundary set skeleton path of P'.
(1) Initialize P";          // Initialize a new path P" in xSG.
// currentInP' is a reference of the current node position in P'.
// First, it is initialized as dst.
(2) currentInP' = dst;
// currentInxSG is a reference to the current node in xSG.
// It is also initialized as dst.
(3) currentInxSG = dst;
// We need a temporary node previous to hold the current position
// in the following while loop.
(4) previous = null;
// Iterate the while loop until currentInxSG is src.
(5) while (currentInxSG != src) {
        // P'.getPreviousNode(node) returns the previous node
        // of node in the path P'.
(6)     previous = P'.getPreviousNode(currentInP');
        // xSG.getEdge(from, to) returns an edge between from and to
        // in the x-Hop sketch graph xSG.
        // It returns null if there is no such edge between them.
(7)     if (xSG.getEdge(previous, currentInxSG) != null) {
            // If the edge is found between previous and currentInxSG,
            // then put previous as the first node of P".
(8)         P".addNodeFirst(previous);
            // Set currentInxSG as previous.
(9)         currentInxSG = previous;
        }
        // For every iteration, move currentInP' backward in P'.
(10)    currentInP' = previous;
    }
```

From lines 6 and 10, the *while* loop proceeds by moving the current position backward by one node in ***P'*** which ensures the *total order* in line 7. Line 7 finds an edge of the two nodes, ***previous*** and ***currentInxSG***, which are h-hops away from each other in the augmented x-Hop sketch graph ***xSG***, where $0 < h < k$. The augmentation for x-Hop sketch graph makes it possible that the algorithm can find an edge between two nodes which are less-than-k-hops away.

Intuitively, the algorithm works as follows. We start from $node_n$. Once we identify $node_n$, we must find $node_i$ ($0 < i < n$) in the augmented k-Hop sketch graph and $node_i$ is an adjacent node of $node_n$. In other words, we must find $node_i$ existing in both P and the k-Hop sketch graph, and one of the edges of $node_i$ in the augmented k-Hop sketch graph must be $node_n$, where $0 < i < n$. The question is whether the edge from $node_i$ to $node_n$ satisfying the above condition exists in the k-Hop sketch graph. Since the number of hops from $node_0$ to $node_n$ is more than k, there must be a node which is k hops or more away to $node_n$, and a k-Hop sketch graph, by definition, has a node k hops away to $node_n$. If $node_i$ is less than k hops from $node_0$, then we have an edge from $node_0$ to $node_i$ because of the sketch graph preparation. If not, we find another node as we did above until we find a node less than k hops from $node_0$.

For example, in Figure 3.8 (b), we cannot find the path from $node_0$ to $node_1$ or $node_2$, but with the augmented 3-Hop sketch graph, we can always find the path, which is analogous to this case. In Figure 3.8 (b), we cannot find the path from $node_0$ to $node_4$ or $node_5$. However, we can with the augmented 3-Hop sketch graph, which is analogous to this case.

3.2.2.2 Calculating the β-approximation

The β-approximation between two nodes has to maintain the following property by definition: the approximation is equal to or more than the actual shortest distance. It is important to have a small difference between the approximation and the actual shortest distance because the smaller the difference is, the more nodes can we prune. One simple way is applying Dijkstra's algorithm on an x-Hop β-sketch graph. That will give us the minimum sum of β-valued edges in the path from a source to a destination. Another way of calculation is applying Dijkstra's algorithm on the same graph with both α- and β-valued edges.

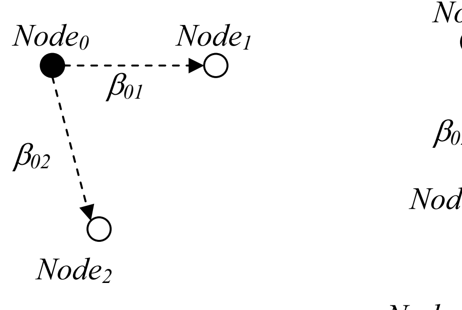
The objective in using the dual-weighted graph is to make the approximation better. To accomplish that, we add one more step to Dijkstra's algorithm. In the usual process, we open adjacent nodes of the node which we are going to close, and each edge of those adjacent nodes has only one value, so the distance of each of those open nodes

will be $c + o$, where c is the distance of the closed node and o is the weight of the edge from the closed node to the open node. Since we use the x-Hop sketch graph with both of the α- and β-valued edges, we have to choose either α- or β-value for each open node to add to the approximation of the closed node. If the approximation of the closed node is determined by $pc + \alpha c$, where pc is the approximation of the predecessor of the closed node and αc is α-value of the edge from the predecessor to the closed node, then we will choose the β-value for the newly opened nodes. Therefore, we use α and β-values alternately along the path. The bottom line for the algorithm is using β-value for the edges of the source node when the path consists of only *one* edge. Other than that, we choose the minimum of the two β-approximations; one starting with β-value for the edges of the source node and the other starting with α-value for the edges of the source node.

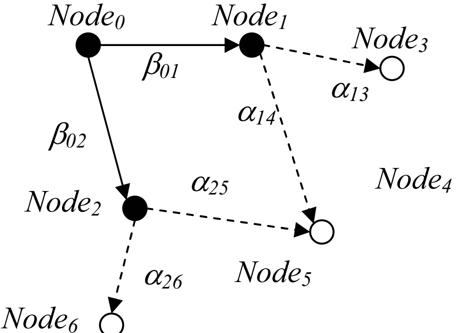
Figure 3.9 illustrates the process. We determine the distance of $Node_2$ by using the β-value (β_{02}). For the distances of its neighbor nodes $Node_5$ and $Node_6$, we use the α-values (α_{25} and α_{26} respectively) to calculate their distances. For the distance of $Node_5$, we have two paths and choose the minimum of (β_{02} + α_{26}) and (β_{01} + α_{14}). In order to open the neighbor nodes of $Node_5$ and $Node_6$, we will use the β-values. We repeat this process until we find the destination.

As in the example, the advantage of using values alternately over Dijkstra's SP algorithm is that we can use the α-value to obtain the β-approximation while we keep the property of the β-approximation.

Algorithm 3.4 shows the pseudo-code for calculating the β-approximations. Line 2 determines whether the approximation starts with α or β. Lines 14 to 31 show the process of choosing either the α-value or β-value. The priority queue used in the algorithm is the same data structure used in Algorithm 3.1.



To open neighbor nodes of $Node_0$, use β-values from $Node_0$ to $Node_1$ and $Node_2$.



To open neighbor nodes of $Node_1$ and $Node_2$, use α-values from those nodes to their neighbors.

Figure 3.9 Dijkstra's Algorithm with Mixed Values

Algorithm 3.4 DijkstraWithDualValue(*src*, *dst*, *dualValuedSG*, $\alpha OR\beta$)

Input: *src* is a source node, and *dst* a destination node. *dualValuedSG* is a sketch graph with both α and β valued edges. $\alpha OR\beta$ is a Boolean value, which determines whether the first value of the approximation starts with α or β.

Output: the approximation distance from *src* to *dst*.

/*Initialize the source node with distance 0*/
1: src.distance ← 0
/*Set true or false for the source node using α or β value for the distance*/
2: *src.is*α ← $\alpha OR\beta$
/*Put the distance information of the source node. *table* is a simple hash map.*/
3: *table.put*(*src*)
/*Initialize the distances of all the nodes in the graph as maximum possible value*/
4: **for** all the nodes *iniNode* in *dualValuedSG* except *src* **do**
  /*Assume *is*α attributes of all the nodes are set to true*/
5:   *table.put*(*iniNode*, **MAX_VALUE**)
6: **end for**
/*Add the source node to the priority queue with distance 0*/
7: *pQueue.enqueue*(*src*, 0)
/*Do this process until the priority queue is empty*/
8: **while** *pQueue.isEmpty*() **do**
  /*Dequeue the minimum distanced node from the priority queue*/
9:   *node* ← *pQueue.dequeue*()
/*If the destination is found, exit the while loop.*/
10:   **if** *node* = *dst* **then exit** the **while** loop
11:   **end if**

```
        /*Get the adjacent nodes of node*/
12:   adjNodes ← dualValuedSG.getAdjacentNodes(node)
13:   for all the nodes adjNode in adjNodes do /*Prepare open nodes*/
        /*If the current node uses α value to calculate the distance from its previous node,
        do the following. This is the only different part from the original Dijkstra's
        algorithm*/
14:     if node.isα is true then
          /*Get the maximum distance from the graph*/
15:       dist ← dualValuedSG.getMaxDistance(node, adjNode) + node.distance
          /*If dist is less than the distance stored in table*/
16:       if dist < table.get(adjNode) then
17:         adjNode.distance ← dist
            /*Set isα as false because the distance calculated for adjNode uses
            maximum distance from node to adjNode.*/
18:         adjNode.isα ← false
19:       end if
20:       table.put(adjNode)
21:       pQueue.enqueue(adjNode, adjNode.distance)
22:     end if
        /*If the current node uses β value to calculate the distance from its previous node,
        do the following. This is the only different part from the original Dijkstra's
        algorithm*/
23:     else if node.isα is false then
        /*Get the minimum distance from the graph*/
24:       dist ← dualValuedSG.getMinDistance(node, adjNode) + node.distance
          /*If dist is less than the distance stored in table*/
25:       if dist < table.get(adjNode) then
26:         adjNode.distance ← dist
            /*Set isα as true because the distance calculated for adjNode uses minimum
            distance from node to adjNode.*/
27:         adjNode.isα ← true
28:       end if
29:       table.put(adjNode)
30:       pQueue.enqueue(adjNode, adjNode.distance)
31:     end else if
32:   end for
33: end while
34: return table.get(dst).distance
```

3.2.2.2.1. Correctness of the β-approximations

The correctness of the process is similar to the argument presented in Section 3.1.3.1. We will prove that the distance of a shortest path from source *src* to destination

dst obtained from Dijkstra's SP algorithm on an augmented *x*-Hop sketch graph with dual-valued edges cannot be less than the actual shortest distance from *src* to *dst*. We have the following cases:

Case 1: A path passes *n* boundary sets, where $n = 1$ or 2.

If $n = 1$, we choose the minimum of,

β-approximation = $maxSD(src, BS_0) + minSD(BS_0, dst)$ *or*

β-approximation = $minSD(src, BS_0) + maxSD(BS_0, dst)$.

If $n = 2$, we choose the minimum of,

β-approximation = $maxSD(src, BS_0) + minSD(BS_0, BS_1) + maxSD(BS_1, dst)$ *or*

β-approximation = $minSD(src, BS_0) + maxSD(BS_0, BS_1) + minSD(BS_1, dst)$.

We already proved similar or the same cases in Section 3.1.3.1.

Case 2: A path passes *n* boundary sets, where $n > 2$.

In that case, we have four possible approximations according to the number of boundary sets and which value the source node takes, and we choose the minimum of them.

β-approximation = $maxSD(src, BS_0) + minSD(BS_0, BS_1) + maxSD(BS_1, BS_2) + \ldots + minSD(BS_{n-3}, BS_{n-2}) + maxSD(BS_{n-2}, BS_{n-1}) + minSD(BS_{n-1}, dst)$ *or*

β-approximation = $maxSD(src, BS_0) + minSD(BS_0, BS_1) + maxSD(BS_1, BS_2) + \ldots + maxSD(BS_{n-3}, BS_{n-2}) + minSD(BS_{n-2}, BS_{n-1}) + maxSD(BS_{n-1}, dst)$ *or*

β-approximation = $minSD(src, BS_0) + maxSD(BS_0, BS_1) + minSD(BS_1, BS_2) + \ldots + minSD(BS_{n-3}, BS_{n-2}) + maxSD(BS_{n-2}, BS_{n-1}) + minSD(BS_{n-1}, dst)$ or

β-approximation = $minSD(src, BS_0) + maxSD(BS_0, BS_1) + minSD(BS_1, BS_2) + \ldots + maxSD(BS_{n-3}, BS_{n-2}) + minSD(BS_{n-2}, BS_{n-1}) + maxSD(BS_{n-1}, dst)$, where BS_i is a boundary set through which the path passes.

In the proof, we show that any of the four approximations cannot be less than the distance of the actual shortest path from *src* to *dst*.

Proof

We consider the first and last nodes in the paths of all the *minSD*s. Let us assume that the nodes of $minSD(BS_i, BS_{i+1})$ are bn_i and bn_{i+1}, where $0 < i < n - 1$. Thus, for i, $minSD(BS_i, BS_{i+1})$ is $minSD(bn_i, bn_{i+1})$. Then, we can derive the following.

β-approximation = $maxSD(src, BS_0) + minSD(BS_0, BS_1) + maxSD(BS_1, BS_2) + \ldots + minSD(BS_{n-3}, BS_{n-2}) + maxSD(BS_{n-2}, BS_{n-1}) + minSD(BS_{n-1}, dst) =$
$maxSD(src, BS_0) + minSD(bn_0, bn_1) + maxSD(BS_1, BS_2) + \ldots + minSD(bn_{n-3}, bn_{n-2}) + maxSD(BS_{n-2}, BS_{n-1}) + minSD(bn_{n-1}, dst)$, and the other three approximations form the similar equations.

By the definitions of *minSD* and *maxSD*, we have the following.

$maxSD(src, BS_0) + minSD(bn_0, bn_1) + maxSD(BS_1, BS_2) + \ldots + minSD(bn_{n-3}, bn_{n-2}) + maxSD(bn_{n-2}, bn_{n-1}) + minSD(bn_{n-1}, dst) \geq$
$maxSD(src, bn_0) + minSD(bn_0, bn_1) + maxSD(bn_1, bn_2) + \ldots + minSD(bn_{n-3}, bn_{n-2}) + maxSD(bn_{n-2}, bn_{n-1}) + minSD(bn_{n-1}, dst) \geq \mathrm{minSD}(src, dst)$.

The other three approximations can be easily proved in the same way. □

Therefore, the β-approximation from *src* to *dst* on the x-Hop sketch graph with dual-valued edges is equal to, or more than, the distance of the actual shortest distance.

3.2.2.3 Calculating the α-approximations

After we determine the β-approximation from source to destination, we calculate the α-approximations from the source to all other nodes in the x-Hop sketch graph, and from all the nodes in the x-Hop sketch graph to the destination. That way, we can determine an α-approximation of a path from the source to the destination, passing an arbitrary node X, by the sum of $m + n'$, where m is α-approximation from the source to X and n' α-approximation from X to the destination. We apply Dijkstra's algorithm on the x-Hop α-sketch graph to calculate the α-approximations. We must be careful to consider in calculating the approximations that the distance must be equal to or less than the actual shortest distance from the source to the destination, passing a certain boundary set which is being probed for pruning.

One thing we have to notice is the α-approximations from all the nodes to the destination in the augmented x-Hop sketch graph, because we have to calculate the α-approximation from each node to the destination, one by one for every node. However, we can solve that problem by building an SP tree from the destination. Therefore, for the α-approximations, we need to build two SP trees, one rooted by the source and the other by the destination. If we want to find the α-approximation from the source to the destination passing X, we first look up the SP tree rooted by the source to find the α-approximation from the source to X, and then look up the SP tree rooted by the destination to find the α-approximation from X to the destination. To make an augmented x-Hop sketch graph for calculating the SP tree *rooted by the source*, we add edges of the boundary sets of the source fragment from h-Hop sketch graphs, where $0 < h < x$. In the same way, we add edges of the boundary sets of the destination fragment from h-Hop sketch graph, where $0 < h < x$, in order to make an augmented x-Hop sketch graph for calculating the SP tree *rooted by the destination*.

3.2.2.3.1. Correctness of the α-approximations

We will prove that Dijkstra's algorithm from A to B on the augmented x-Hop α-sketch graph finds a shortest path whose distance is equal to or less than the actual shortest path from A to B. We will use the *lemmas* introduced in Section 3.2.2.1.1.

Lemma 3.6: The distance of the path found by Dijkstra's algorithm from A to B on the augmented x-Hop α-sketch graph is equal to, or less than the distance of the shortest path from A to B in the graph.

Proof

Let P be the boundary set skeleton path of the shortest path *SP* from *src* to *dst* in the original sketch graph, where *src* is the source node and *dst* the destination node. By *lemmas* 3.4 and 3.5, we can convert P into P' which is the simplified boundary set skeleton path of the shortest path in the augmented x-Hop α-sketch graph of the original sketch graph. Then, the distance of P' is

(1) $minSD(src, BS_0) + minSD(BS_0, BS_1) + minSD(BS_1, BS_2) + \ldots + minSD(BS_{n-1}, dst)$.

The boundary sets in (1) are also the boundary sets through which the actual shortest path passes. Therefore, we can assume the boundary nodes represented by those boundary sets are bn_0, $bn_1 \ldots bn_{n-1}$, which are also the nodes in *SP*. Then, we have the following by the definition of *minSD*, $minSD(BS_i, BS_{i+1}) \leq minSD(bn_i, bn_{i+1})$, where $0 < i < n - 1$.

(2) The distance of $SP = minSD(src, bn_0) + minSD(bn_0, bn_1) + minSD(bn_1, bn_2) + \ldots + minSD(bn_{n-1}, dst) \leq minSD(src, BS_0) + minSD(BS_0, BS_1) + minSD(BS_1, BS_2) + \ldots + minSD(BS_{n-1}, dst)$.

(2) shows that the distance of *P'* is equal to, or less than that of the shortest path. Dijkstra's algorithm on the augmented *x*-Hop α-sketch graph finds the shortest path of all possible paths including *P'*. Let us assume that P_{Dijk} is the path found by Dijkstra's algorithm on the augmented *x*-Hop α-sketch graph. There are two possible cases.

Case 1: $P_{Dijk} = P'$.

In case 1, we already proved that the distance of P' is equal to, or less than the one of the shortest distance.

Case 2: $P_{Dijk} \neq P'$.

In case 2, P_{Dijk} cannot be more than *P'* because if P_{Dijk} were more than *P'*, then Dijkstra's algorithm would find *P'* as the shortest path and that would be a contradiction. Therefore, $Dist(P_{Dijk}) \leq Dist(P') \leq Dist(SP)$, where *Dist*(*Path*) is a function returning the shortest distance of *Path*.

Therefore, the distance of *P'* is equal to, or less than the distance of *SP*.□

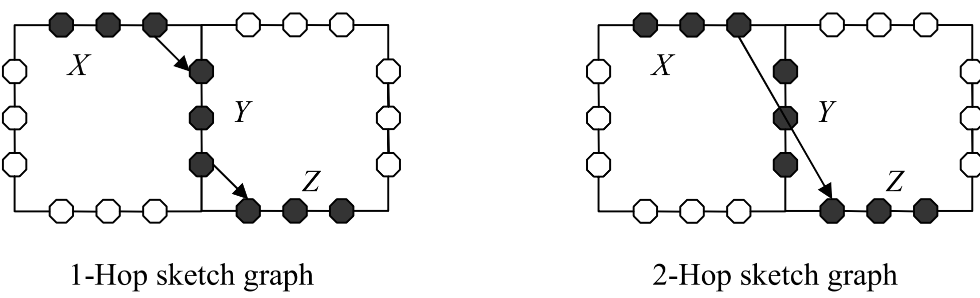
3.2.2.4 Pruning Boundary Sets

The last step of the pruning algorithm is to process every boundary set if it is eligible to be pruned. A boundary set *X*, a node in the sketch graph, is pruned if and only if

- SPT_S(*source*, *X*) + SPT_D(*X*, *destination*) > the β-approximation, where SPT_S and SPT_D represent the shortest-path trees rooted by the source and the destination respectively on an augmented *x*-Hop sketch graph, and *SPT*(*A*, *B*) is a function returning the shortest distance from *A* to *B* in the shortest-path tree.

If the above condition is satisfied, we can safely eliminate the boundary set *X*. The above statement is the same as discussed and proved in Section 3.1.3.3, so we will skip the proof here.

In using an *x*-Hop sketch graph, the larger *x* is, the more accurate approximations we can get. For example, let us assume we know the α- and β-values from the boundary set *X* to the boundary set *Y*, from *Y* to the boundary set *Z*, and from *X* to *Z*. The α- and β-values from *X* to *Y* and from *Y* to *Z* can be considered as values of edges in a *1*-Hop sketch graph and the values from *X* to *Z* as ones in a *2*-Hop sketch graph. It is obvious that the sum of the α-values of edges from *X* to *Y* and from *Y* to *Z* is equal to, or smaller than the one from *X* to *Z* (Figure 3.10). Since we want to get as large a value as possible for the approximation for the lower bound, we should use an *x*-Hop sketch graph with a larger *x*. For the approximation of the upper bound, the same argument is applied.



The graph on the left depicts a 1-Hop sketch graph, with 1-hop edges from boundary set *X* to *Y*, to *Z*.

The graph on the right depicts a 2-Hop sketch graph, with 2-hop edge from boundary set *X* to *Z*, which is 2-hop because it is passing boundary set *Y*.

Figure 3.10 Difference Between 1-Hop and 2-Hop Sketch Graph

After finishing the pruning process, we will use the only boundary sets which have survived pruning to find the shortest path in the disk-based SP algorithm.

Algorithm 3.5 shows the pseudo-code for the pruning process. Lines 2 to 8 calculate the β-approximation by choosing the minimum of the two possible β-approximations. Lines 9 to 15 show the steps of pruning the boundary sets. Lines 9 and 10 build the shortest path trees rooted by the source and destination on the augmented x-Hop α-sketch graph.

Algorithm 3.5 X-HopSketchGraphPrune (*sg*, *xHopSG*[], *src, dst, srcF, dstF*)

Input: *sg* is an original sketch graph. *xHopSG* is an array containing a 1… x-Hop sketch graph, where x is the number of hops, *src* a source node, and *dst* a destination node. *srcF* is a fragment where *src* is, and *dstF* is a fragment where *dst* is.

Output: Pruned *sg*.

/*Convert x-Hop Sketch Graph for pruning process.*/
1: *xHopSG*[*xHopSG.length* – 1] ← *MakeAugmentedXHopSG*(*xHopSG*, *src*, *dst*, *srcF*, *dstF*)
/*Calculate the β-approximation starting with *maxSD**/
2: *thePivot_0* ← *DijkstraWithDualValue*(src, dst, *xHopSG*[*xHopSG.length* – 1], **TRUE**)
/*Calculate the β-approximation starting with *minSD**/
3: *thePivot_1* ← *DijkstraWithDualValue*(src, dst, *xHopSG*[*xHopSG.length* – 1], **FALSE**)
/* Choose the minimum of *thePivot_0* and *thePivot_1*.
4: **if** *thePivot_0* < *thePivot_1* **then**
5: *thePivot* ← *thePivot_0*
6: **end if**
7: **else** *thePivot* ← *thePivot_1*
8: **else end**
/*Build the shortest path tree rooted from *src* using α sketch graph*/
9: $SPT_{SxHopSG}$ ← *Dijkstra*(src, *xHopSG*[*xHopSG.length* – 1])
/*Build the shortest path tree rooted from *dst* using α sketch graph*/
10: $SPT_{DxHopSG}$ ← *Dijkstra*(dst, *xHopSG*[*xHopSG.length* – 1])
11: **for** all the nodes *node* in *sg* **do** /*Pruning process starts*/
/*if the sum of two approximations for the lower bound is more than the approximation for the upper bound*/
12: **if** thePivot < ($SPT_{SxHopSG}$.*getMin*(*node*) + $SPT_{DxHopSG}$.*getMin*(*node*)) **then**
13: *sg.remove*(*node*) /*Remove the node*/
14: **end if**
15: **end for**
16: **return** *sg*

3.3 Query Optimization Using Query Graph

The problem we face when multiple queries come into the spatial database system is how to sort the queries into a new order so that we can compute them with minimal system resources. We assume that a query usually comes into the system with the following information: a source, a destination coordinate, and fragments in which the source and destination coordinate lie. As we have seen in Section 2.2, the first part of finding the shortest path, namely finding the *skeleton path* of a query, requires those two fragments in main memory. Another assumption is that we have only enough buffers to accommodate *two* fragments at a given moment. In addition, the strategy for managing buffers in this thesis is the *Least Recently Used* (*LRU*), which is the most widely used in modern computer systems. Under these constraints, we can easily calculate how many swaps of fragments are required to process queries. For example, we have two different schedules of three queries pending in the queue, as illustrated in Figure 3.11 (a). In Figure 3.11 (b), the optimal schedule of processing those queries requires 4 I/O activities in the *LRU* buffer: reading in *fragment 0* and *1* for $Query_0$, reading *fragment 2* for $Query_3$, and reading *fragment 3* for $Query_2$. On the other hand, the poor schedule in Figure 3.11 (c) requires 6 I/O activities: 2 reads of the fragments for each query respectively. In order to get the optimal schedule for *n* queries, the expected calculation time is *n!*, which is non-polynomial. Therefore, it is impossible to get the optimal schedule for *n* queries within a reasonable time if *n* is large enough.

If we think of the IDs of the source and the destination fragments as nodes in a graph and draw a line between the two nodes, we can consider the above problem to be a graph traversal problem, which is known as an NP-complete problem to get an optimal result for visiting each node as little as possible [14]. Since it is an NP-complete problem, we would rather try to find a heuristic algorithm which could lead to a near optimum schedule. The heuristic algorithm we are proposing here is very simple and fast, but it is effective. The test results of the algorithm will be given in a later chapter.

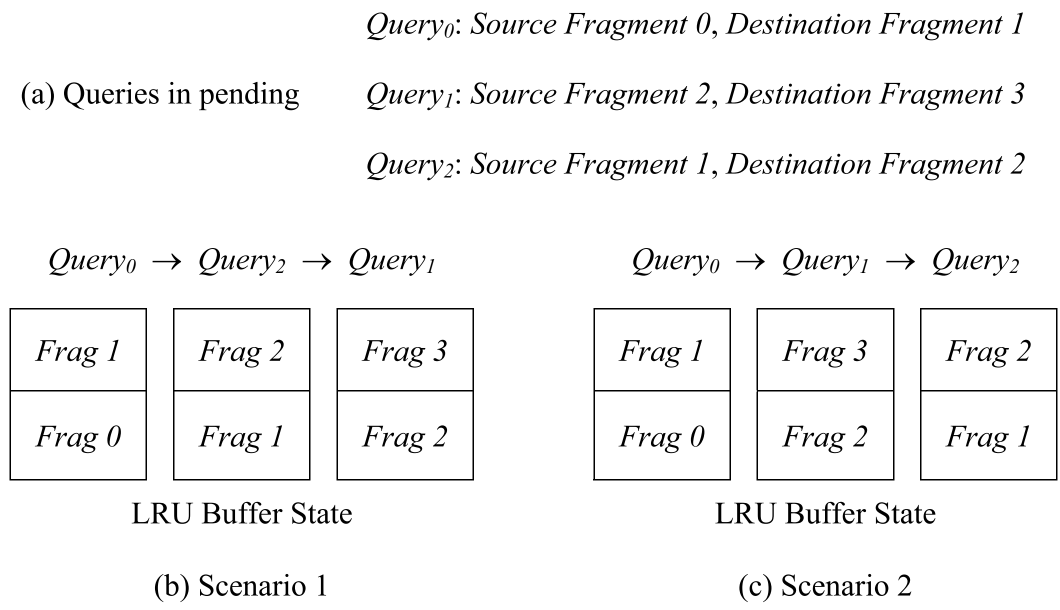


Figure 3.11 Query Optimization

One of our objectives in devising the algorithm is that the query-scheduling algorithm should not affect the overall performance of the shortest path query, which means it should not take more than a few seconds for a large number of queries. To achieve the above, we have found an algorithm with an execution cost of $O(n)$.

Before describing the algorithm, we need to clarify a simple step of grouping queries. If two or more queries in the query set have the same source and destination fragments, the buffer does not need to swap its contents in order to calculate those queries. We call them an *Equivalent Class* (*EC*) of queries. Hence, an *EC* consists of queries sharing the same source and destination fragments. In fact, we loosen the definition of *EC* as we consider queries to be *EC* if they have the same fragments for the source and destination fragments in either order. For example, one query with fragment ID *1* for the source and fragment ID *2* for the destination, and the other one with fragment ID *2* for the source and fragment ID *1* for the destination are *EC* in the loose sense.

Algorithm 3.6, *QueryGraph*, describes how we build a query graph for queries and sort them. The advantages of the algorithm are that it is simple and efficient, and

does not need any complicated data structure. For the algorithm, we assume the buffer size for the fragments is 2. We also need to define a few terms we are using for the algorithm. A node is *isolated* if it is not connected to any edge in a graph. The node u is called a *terminal* node if the degree of u is 1. An edge $e = <n, u>$ is called a *dangling edge* incident to n if u is terminal. *Processing an edge* is defined as outputting the edge then removing the edge and removing any isolated nodes from the graph. The algorithm basically examines the graph and removes edges and nodes step by step. Edges in the graph denote queries, so when edges are removed, the queries denoted by the edges are stored in a new sequence of the query set. Therefore, after removing all the edges in G, the algorithm will generate a new query sequence with new ordering, possibly reducing I/O activities in the buffer.

The complexity of the algorithm is $O(n)$ since there is only one single loop. Inside the loop, there is not much calculation, simply picking a random node, or terminal node, and then removing possible *dangling edges* attached to the current node *cn*. The only redundant calculation in the algorithm is finding *dangling edges* attached to *cn* because *non-dangling* edges will be processed every time. However, the cost for that is limited since finding *dangling* edges is trivial, which only requires checking the nodes of the edges attached to *cn*.

The highlight of the algorithm is *processing dangling edges*. Since *dangling edges* guarantee there will be at least one fragment in the buffer, the algorithm maximizes the usage of the current contents in the buffer. In Chapter 4, we will show how much the algorithm improves the buffer utilization compared to non-scheduled queries.

Algorithm 3.6 *QueryGraph*

Input: ECs and a query graph *graph*.

Output: ECs with new order

/* initialization */
1: *cn* = null
/* newly ordered queries will be stored in the *query queue* */
2: *query queue* is created.
/*do the loop until all the nodes in the graph are removed */

```
3:  while graph != empty do
4:    if (cn == null) then
5:      if (there is an edge e = <u, v> such that
        v is terminal and the degree of u is either one or two) then cn = v;
6:      end if
7:      else cn = w, where w is any node in graph.
8:      end else
9:    end if
10:   if (there are dangling edges incident to cn) then
11:     process the dangling edges one by one;
12:   end if
13:   if (cn exists) then
14:     let e = <cn, v> be an edge incident to cn;
15:     process e;
16:     cn = v, if v exists and null otherwise;
17:   end if
18:   else cn = null
19:   end if
20: end while
```

3.4 Shortest-Path Algorithm – Batch Disk-based SP Algorithm

The three algorithms introduced earlier in this chapter focus on how to organize a set of queries and to minimize search spaces. They do not themselves find the shortest path; they prepare queries for the better performance of the shortest-path algorithm on partitioned graphs.

The disk-based shortest path algorithm in [7] has two steps: finding a skeleton shortest path and filling the skeleton path. Since a skeleton path consists of boundary nodes in the path from the source to the destination, the next phase of the algorithm is to fill out the intermediate nodes between any two consecutive boundary node pairs in the skeleton path. Those two parts complete the algorithm. To fill out those intermediate nodes, we apply Dijkstra's SP algorithm on a fragment in which two consecutive boundary nodes in the sketch graph lie. Since fragment DB is normally big, their solution is to place those fragments in the external memory and load them when necessary.

If there are multiple queries pending in the system, a little modification of the order of processing multiple queries would help the algorithm use fewer I/O activities. Instead of processing them one by one, a batch disk-based SP algorithm process as a

certain number of queries together to find skeleton paths and then fill out those skeleton paths together at a later stage. The scheme is very simple. First, we calculate the skeleton paths of queries and then identify the fragments needed to compute the partial shortest paths of the queries. After we know all the fragments, we calculate the partial paths with respect to the fragments of the partial shortest paths. The last step is simply to place the calculated partial paths into the proper positions in the skeleton paths.

Figure 3.12 is an example of the processing of two queries. The shortest path of the first query is to pass through fragments 0, 2, 3, 5, 6, and 7 after the computation of the skeleton path. The shortest path of the second query passes through fragments 1, 2, 4, 5, 6, 8, and 9. After the computation of the skeleton paths of each query, we have to fill the skeleton paths. To complete the computations, for query 1, we need to apply Dijkstra's algorithm on 6, 5, 3, and 2; for the query 2, we need to apply it on 8, 6, 5, 4, and 2. If we have a buffer size of 2 and process query 1 first, and then query 2, we need 9 reads of the fragments to fill out the skeleton paths of the two queries. If we consecutively compute the partial shortest paths for the same fragment from different queries, we can save I/O activities. In the example, fragments 2, 5, and 6 are necessary to fill both skeleton paths, so we calculate the partial shortest paths for the two queries together for those fragments. This strategy allows us to save 3 reads of fragments.

This strategy works better if queries are scheduled properly. The suggested algorithm in Section 3.3 is effective since it sorts queries by their locality.

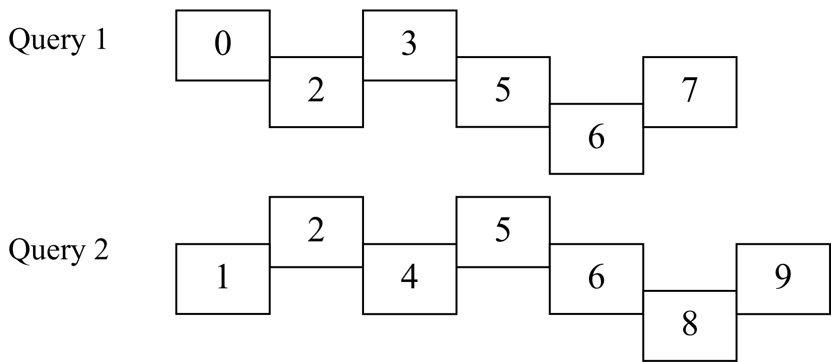


Figure 3.12 Accessed Fragments

3.5 An Example of the Algorithms

This Section will give an example of the algorithms presented in previous sections, from the start to the end of processing one query. For the pruning part, the example uses *BSDistMatrix* instead of an *x*-Hop sketch graph. The only difference between the two lies in how to calculate approximations, and the explanation of calculating with an x-Hop sketch graph is given in its own section. We assume that we already have a partitioned graph, the boundary node distance matrix, which holds all the shortest-distance information of boundary nodes in each fragment, and the *BSDistMatrix*, which holds all the shortest-distance information between boundary sets in a partitioned graph. Those are outcomes of the pre-processing phase of the disk-based *SP* algorithm.

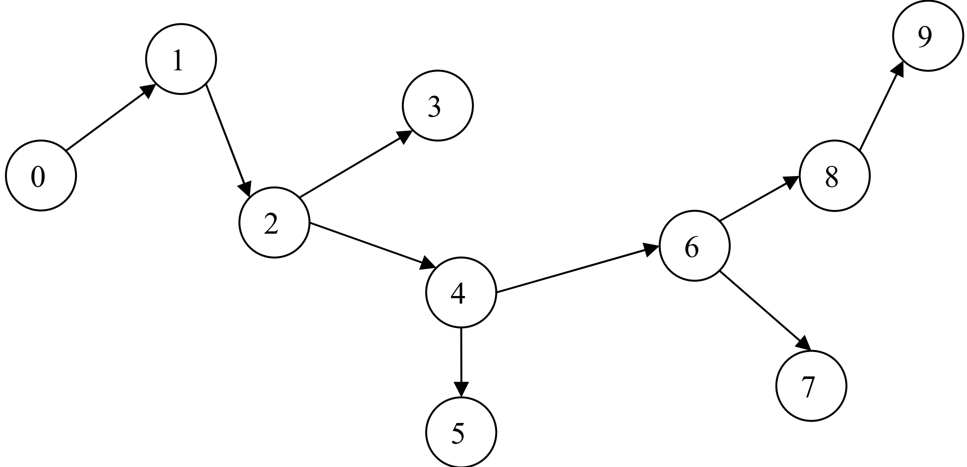


Figure 3.13 Query Graph

The first step of the query process is to accept queries and group them into batches. After grouping, the program sorts each group so that queries in each group will read in a fragment database.

Let us assume we have the queries depicted in Figure 3.13, and we have only a cache size of 2 for the fragment database. Each node represents a fragment. A node with the beginning of an arrow indicates the source fragment, while a node with the ending of an arrow indicates the destination fragment. For example, fragment 2 can be a destination fragment with fragment 1 and a source fragment with fragments 3 and 4. As described in

Section 3.3, sorting those queries is simple. After constructing a query graph like Figure 3.13, we must randomly choose any node in the query graph. We assume that we select fragment 2. With the selected fragment, we must go in the direction of the arrows. Fragment 2 has two possible routes, but the arrow between fragments 2 and 3 is a *dangling edge*, so we remove that arrow first. The next node will be 4; therefore, we remove the arrow from fragments 2 to 4. Processing fragment 4, we find another *dangling edge*, the arrow between fragments 4 and 5. We remove it first before going to fragment 6. The possible result will be 2 → $\underline{3}$ → 4 → $\underline{5}$ → 6 → $\underline{7}$ → 8 → $\underline{9}$ → *1* → $\underline{0}$. The underlined numbers indicate fragments with *dangling edges*.

Once the sorting of queries in a group is completed, we then find the skeleton path for each query individually. Finding a skeleton path of a query merely involves applying Dijkstra's SP algorithm using a partitioned graph and its auxiliary files, such as the distance matrix, instead of a normal graph. In addition, we are going to insert the pruning algorithm into the middle of the algorithm.

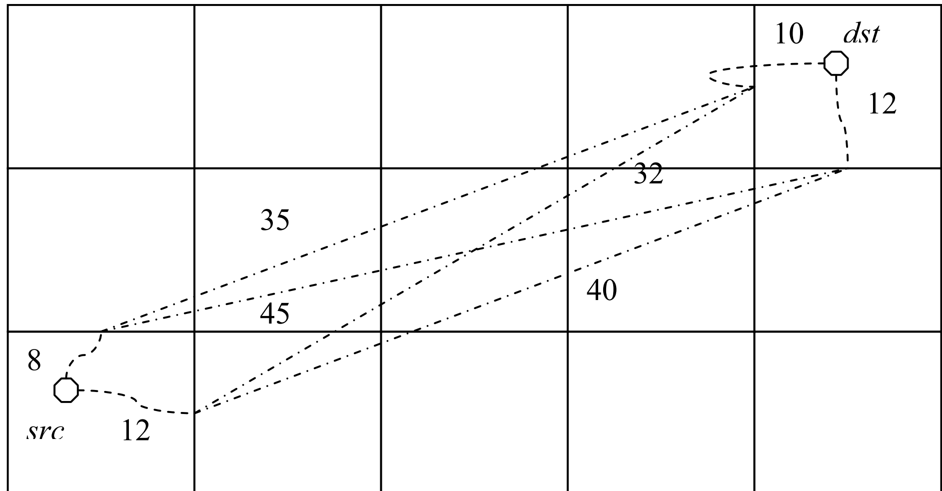
As described in Chapter 2, we need source and destination fragments, and the distance matrix to carry out the algorithm. The pruning process starts with calculating the shortest-path tree rooted at the source in the source fragment and another shortest-path tree rooted at the destination in the destination fragment. With those shortest path trees, we are able to calculate the approximations with *BSDistMatrix*, taking the minimum approximation from the possible approximations. There can be ($2 \times m \times n$) approximations, where m and n are the number of boundary sets in the source and destination fragments respectively. Each approximation is the minimum of the sums of 1) the maximum (minimum) shortest distance from the source node to a boundary set in the source fragment, 2) the minimum (maximum) shortest distance between the boundary set in the source fragment to a boundary set in the destination fragment, and 3) the maximum (minimum) distance from the boundary set in the destination fragment to the destination node respectively (Section 3.1.2). After calculating the approximation, we can compare it with the minimum distances passing other boundary sets in the partitioned graph. A minimum distance passing a boundary set X is selected out of a number of possible minimum distances. There can also be $m \times n$ possible minimum distances, the same as

above. Each possible minimum distance is the sum of 1) the minimum shortest distance from the source node to a boundary set in the source fragment, 2) the minimum shortest distance from the boundary set to boundary set X, and 3) the minimum shortest distance from boundary X to a boundary set in the destination fragment. If the minimum distance passing X is longer than the approximate shortest distance, then we can safely remove X, which means we do not have to consider the boundary nodes in X during the remaining part of finding a skeleton path. The remaining part of the process just follows the normal procedure of the disk-based SP path algorithm. The figures from 3.14 to 3.16 depict an example of finding the skeleton path in the pruning process. Figure 3.14 shows how to calculate the β-approximation from the source to the destination. For the sake of simplicity, the figure only shows the (*max*– *min* – *max*) approach. Figure 3.15 explains how to calculate the α-approximation and prune a boundary set in a graph. The figure shows the case of probing the boundary set X, whose distance is longer than the β-approximation calculated in Figure 3.14. Therefore, X can be pruned. We probe every boundary set in the figure, in order to decide whether it can be pruned or not. Figure 3.16 shows finding the skeleton path of the query on the pruned graph. We assume that grey-coloured boundary nodes are only eligible for the calculation, which means all other boundary nodes are pruned. Therefore, the search space for the disk-based SP algorithm becomes smaller with the pruning algorithm.

After the pruning algorithm, we apply the batch disk-based SP algorithm, on the pruned graph. Since the skeleton path consists only of boundary nodes, we complete the path by filling out intermediate nodes between boundary nodes in the skeleton path, using the traditional Dijkstra's SP algorithm. In the batch disk-based SP algorithm with multiple queries, the filling-out process is slightly different from the disk-based SP algorithm. Since we know which fragments are required to be filled out, we can fill the skeleton paths in each fragment. Figure 3.17 shows an example of two queries filling out the skeleton paths. If we have only two cache entries for the fragment database and process each query one by one, then we need 8 reads of the fragment database. However, if we group the partial shortest paths by their fragments as in Figure 3.17, we need only 5

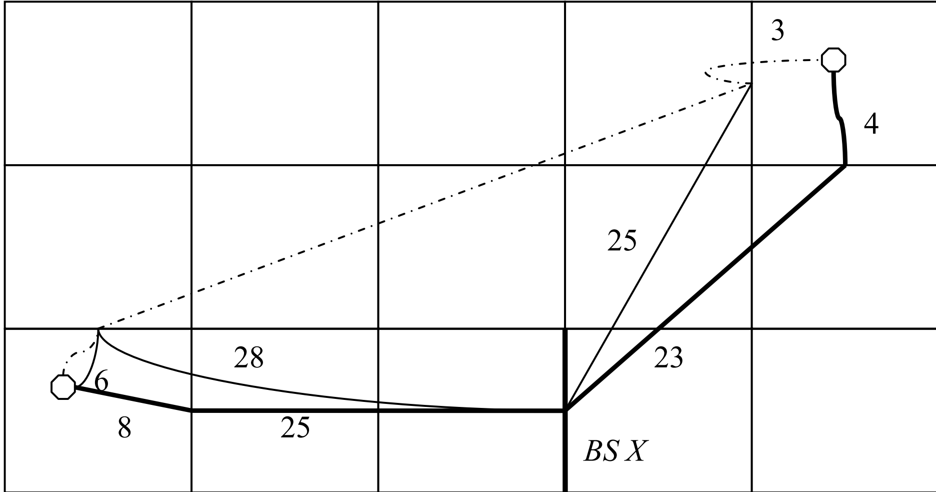
reads since 3 fragments overlap and we process the partial shortest paths for those fragments together.

In sum, the algorithm sorts out multiple queries so that we can minimize the fragment database access. The algorithm then processes each query to find the skeleton path. During the process of finding the skeleton path, once the algorithm gathers the information of minimum and maximum distances from the source to the boundary sets in the source fragment, the pruning algorithm is activated. After the pruning, the normal disk-based SP algorithm is applied on the pruned graph. In the filling-out process, the algorithm groups skeleton paths by fragments and then fills out each partial skeleton path.



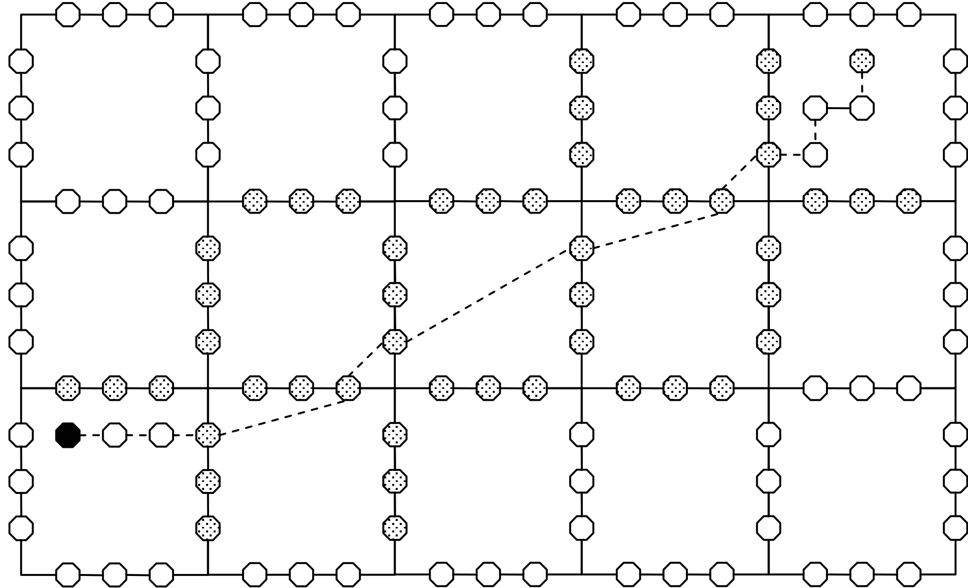
1. We have the maximum shortest distances from *src* and *dst* to the boundary sets in their fragments.
2. To calculate the β-approximations, we need to access *BSDistMatrix*, and get the minimum distances between boundary sets in the source and destination fragments.
3. Since there are two boundary sets in each fragment, we have 4 possible approximations. Select the minimum from them. In the example, we have 53 (8 + 35 + 10) is the minimum of all 4 candidates.

Figure 3.14 β-approximations



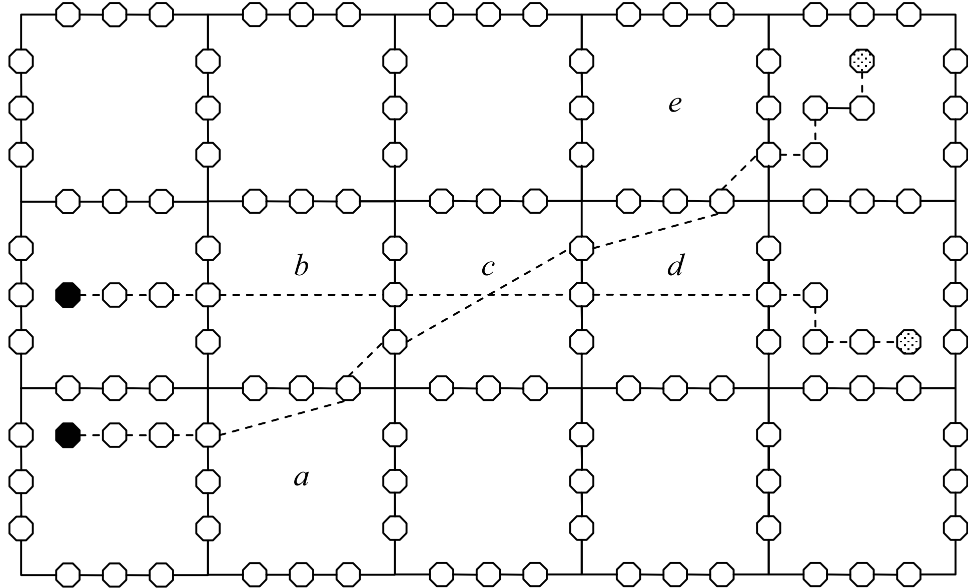
1. The dotted line represents the β-approximation from the source to the destination, with the distance of 53.
2. The solid lines in the source (destination) fragment represent the minimum shortest distances from the source (destination) to the boundary set in the source (destination) fragment.
3. We now compare the β-approximation to other α-approximations passing a boundary set, in this example, *X*.
4. There are also 4 possible candidates for the approximation passing the boundary set *X*, represented by solid lines.
5. We calculate those candidates and select the minimum distance, 60 (8 + 25 + 23 + 4).
6. We remove the boundary set *X* since the α-approximation (60) is larger than the β-approximation (53).

Figire 3.15 Pruning a Boundary Set



1. After pruning, we only need to consider the grey boundary nodes.
2. With the pruned graph, we now have the skeleton path, represented by solid line.

Figure 3.16 Finding the Shortest Path with Pruned Graph



1. We have two skeleton paths.
2. The first query passes through a, b, c, d, and e, and the second query passes b, c, and d.
3. Both skeleton paths pass the grey coloured fragments, b, c, and d commonly, which means we can fill out the partial shortest paths in those fragments together.

Figure 3.17 Dealing with Multiple Queries

Chapter 4
Experiments

The main purpose of chapter 4 is to detail how the algorithms presented in this paper perform, compared to Dijkstra's algorithm and the disk-based algorithm proposed by [7]. We divided the performance testing into three sections: QueryGraph (Section 3.3), the batch disk-based algorithm (Section 3.4), and pruning algorithms (Sections 3.1 and 3.2).

4.1 System Environment and Data Sets

The system for testing is a Pentium 4 1.7GHz with 256MB of main memory. The hard disk of the system is Ultra ATA/100, with a 7,200 rpm spinning rate. Java is the primary language, and the version is 1.3.1. To make a homogeneous environment for every test case, we set the Java Virtual Memory (JVM) to be 128MB, which means the total memory we use for the test is 128MB.

The data for testing is from the Connecticut road system extracted from the Tiger/Line file [15]. When the Connecticut road system is represented as a graph, the file size is about 20MB. It consists of around 190,000 edges and 160,000 nodes. To partition the graph, we adopt the partitioning algorithm in [7]. The details of the partitioned graph and its auxiliary data files are summarized in Table 4.1. We use different fragment sizes, ranging from 100 nodes to 15,000 nodes per fragment, to see how fragment size impacts the performances.

As shown in Table 4.1, the number of nodes per fragment does not affect the size of the DB files, except that of *BSDistMatrix*. *BSDistMatrix* is a 2-dimensional matrix, having rows and columns for boundary sets. Therefore, the file size of *BSDistMatrix* is proportional to the square of the number of the boundary sets in each test set. The time to build *BSDistMatrix* is different in each case because the process of building *BSDistMatrix* is basically calculating the shortest paths between all possible pairs of boundary nodes. For example, the case of 1,000 nodes per fragment takes about 4 times longer than the one of 15,000 nodes per fragment, which is almost the same ratio as the number of boundary nodes between the two cases.

No. of Nodes per fragment	No. of fragments	No. of boundary sets	No. of boundary nodes	Fragment DB size (MB)	Distance Matrix DB size (MB)	*BSDistMatrix* size (MB)
All	1	-	-	18.9	-	-
100	1693	3430	12251	23.4	2.76	185.221
1,000	138	347	3998	20.0	2.20	2.023
5,000	28	66	1649	19.4	1.72	0.09
10,000	14	29	1182	19.3	1.72	0.020
15,000	10	21	1120	19.3	2.04	0.013

Table 4.1 Test Set Statistics

Table 4.2 shows the file size of the *x*-Hop sketch graph for fragments having 1,000 and 5,000 nodes. The reason that the file size of the graphs peaks at the 5-Hop sketch graph in the 1,000 node fragment is that some of the nodes in the sketch graph do not have nodes that are 6-hops away. The size of the graphs in the 5,000-node fragment does not grow any more after a 7-Hop sketch graph, which means the maximum number of hops in the sketch graph between any given two nodes does not exceed 7.

No. Of Nodes	$x = 1$	$x = 2$	$x = 3$	$x = 4$	$x = 5$	$x = 6$	$x = 7$	$x = 8$	$x = 9$	$x = 10$
1000	0.395	0.562	0.676	0.745	0.772	0.768	0.735	0.664	0.589	0.519
5000	0.127	0.147	0.147	0.136	0.125	0.117	0.113	0.113	0.113	0.113

Table 4.2 The Size (MB) of x-Hop Sketch Graphs

The time it takes to build a *BSDistMatrix* for a 1,000-node fragment is about 8,000 seconds using the system. Generating all x-Hop sketch graphs, where $0 < x \leq 10$, for a 1,000-node fragment, takes about 5,000 seconds. As we expected, making x-Hop sketch graphs at a certain level takes less time than making a *BSDistMatrix*.

For the partitioned graph (i.e. fragments), we need less than 60MB of memory with cache size management. To control the cache size for the databases of partitioned graphs, we employed the VirtualHashtable technique introduced in [7]. The VirtualHashtable is an array-like data structure which reads in and writes out memory contents to the hard disk. Its key feature is that not all of its content is in the main memory. The user specifies the maximum amount of content which the VirtualHashtable can hold. Therefore, if the VirtualHashtable tries to load content while it already has the maximum amount of content, then it writes some of the current content out to disk and loads the new content by means of the LRU replacement scheme. In the following test results, all the cache sizes being set up denote the maximum amount of content in the VirtualHashtable.

During the test, the most memory-consuming data bases are the fragment DB and the distance matrix DB. *BSDistMatrix* is used at most once throughout the process of each query; however, the fragment DB and the distance matrix DB are used at least once throughout the computation of the shortest path, so the cache sizes of those DBs are critical to the outcomes. Even if we have the JVM of 128MB for the test, we try to set the cache size as low as possible to fit into 64MB, so that we can assume that the suggested algorithms are scalable for larger graphs, which cannot be loaded in the main memory if they are not partitioned.

4.2 Query Optimization Using Query Graph

Algorithm 3.1, *QueryGraph*, tries to minimize the I/O activities for the fragment DB. There are two places where the fragment DB is necessary during the shortest-path calculation: in finding the skeleton path and filling it out. The usefulness of *QueryGraph* is for the stage of finding skeleton paths to read the source and destination fragments from fragment DB as little as possible. The assumption is a cache size of 2, as mentioned in Section 3.3, and that all the queries are pre-sorted to equivalent classes. With only 2 cache entries, the optimized schedule will be able to re-use a maximum of 50% of the cache if the query graph of the equivalent classes is all connected, because, in the optimized schedule, there will always be one cache entry for the next query to use. The worst case is 0% cache utilization.

For the test, we randomly generated 10,000 queries and then divided them into small queues of specific sizes. The queues are the batches we process at the same time. For example, if the size of a queue is 10, we use *QueryGraph* on queries in the queue and calculate their shortest paths together.

The result of the test is obtained by executing only *QueryGraph* with the pre-sorted equivalent classes of the queries. Therefore, the test is independent of all other phases.

No. of Queries in the Queue	Non-scheduled	10	20	50	100	1000
Cache Utilization	0.0144	0.047	0.120	0.223	0.343	0.471

Table 4.3 Cache Utilization of Using *QueryGraph* Algorithm

Table 4.3 shows how much *QueryGraph* improves the cache utilization with respect to the size of the queries in the queue. The cache utilization is calculated by p / q, where p is the number of cache-hits and q the number of total requests for the fragment DB. It is obvious that more queries in a batch will increase the cache utilization since there will be a greater possibility of sharing nodes when a large number of queries are in the batch. Even a case of 10 queries in the queue performs more than 3 times better than a

non-scheduled case. A case of 1000 queries in the queue reaches near the maximum utilization level, that is 50% of the optimized schedule, while the time to schedule takes less than 0.1 seconds.

4.3 Disk-based SP Algorithm vs. Batch Disk-based SP Algorithm

The difference between the disk-based SP algorithm and the batch disk-based SP algorithm lies in how to process multiple queries. The disk-based SP algorithm executes queries one by one, which means that there is no interruption between queries. On the other hand, the batch disk-based SP algorithm has two steps to process a unit of queries. First, it calculates skeleton paths for all queries in the batch. With the calculated skeleton paths, we know which fragment we need to read for filling-out process, so the batch disk-based SP algorithm fills the skeleton paths by fragment (Section 3.4). Since the purpose of the batch disk-based algorithm is to reduce the I/O activity for reading the fragment DB during the filling-out phase, it should not affect the finding-skeleton-path phase and is, in fact, implemented so as not to affect it.

In this section, we group 10 queries together and calculate them by means of the two algorithms above. A unit of 10 queries is also scheduled by the *QueryGraph* algorithm. First, we calculate 300 queries sequentially using the disk-based algorithm, and then count the number of requests for the fragment DB. We then calculate 300 queries, grouped 10 at a time. The skeleton paths of the 10 queries are calculated individually, and the filling-out process for them is carried out together. For the same reason as shown in Section 4.2, the more queries that are processed together, the more benefits we get. The cache size of the fragment DB for testing is 2.

The result of the test is obtained by executing two algorithms separately. Both algorithms use *QueryGraph* to sort the queries. *QueryGraph* does not affect the result of the test because the sequence of queries for calculating skeleton paths remains the same regardless of the two algorithms. The pruning algorithms are not used in this test. The cache size for the distance-matrix DB is set to the maximum so that the algorithms are not affected by the cache size.

Figure 4.1 shows that the batch disk-based algorithm requests around 20% fewer fragment DB accesses than the disk-based SP algorithm does. As the fragment size increases, the number of fragment-DB accesses decreases, which increases the possibility of overlapping fragments between different queries. The fewer fragment requests during calculations also affect the calculation time.

The savings in terms of calculation time are also one of the benefits of the algorithm. Our results show that the calculation time decreases by up to 20% in the best case with 10-query grouping. The details of the results will be shown in Section 4.4.5.

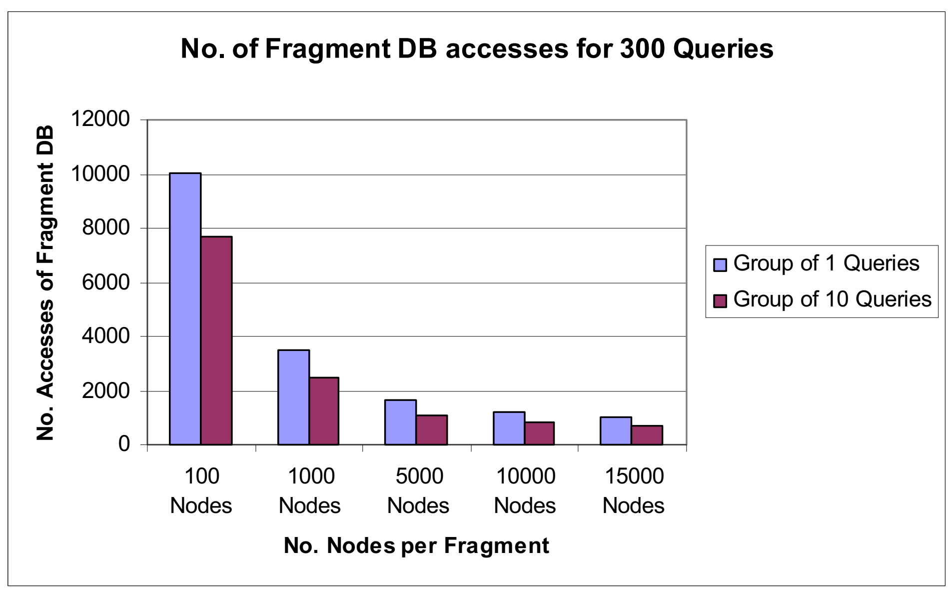


Figure 4.1 Number of Fragment DB Accesses

4.4 Performance with Pruning Algorithms

We introduce two different pruning methods, one using *BSDistMatrix* and the other an x-Hop sketch graph. For a consistent testing environment, we first find the optimum parameters for some important factors. The factors for the test are as follows:

- Query Type: We divide queries into three types of ranges: long, medium, and short. The shorter query might benefit more from the pruning algorithm, while the longer query might benefit less, in terms of the number of pruned boundary sets. Long-range queries are more than 66% of the longest possible

distance in the graph, medium-range queries are less than 66% and more than 33%, and short-range queries are less than 33%. We will carry out all the testing according to the differently sized sets of queries.

- Fragment Size: Fragment size matters because, in partitioning a graph, we have fewer fragments if we set the number of nodes in a fragment to a large number, which in turn means we have a smaller number of boundary sets in the partitioned graph. The number of boundary sets affects two aspects. One is the size of materialized data because the size of materialized data increases as the boundary sets grow. The other one is the effectiveness of the pruning algorithm. Too many boundary sets take too much time to read in the data, as well as more time to calculate. We test 5 different fragment sizes: 100, 1,000, 5,000, 10,000, and 15,000.
- Cache size of Distance Matrix: As proved in [7], the cache size is set for the best performance at some level smaller than the full cache size. For consistent testing results, all tests should have standardized cache sizes. We test 5 different levels of cache sizes.
- Degree of an x-Hop sketch graph: In an x-Hop sketch graph, x is an important factor since the accuracy of the approximations changes according to x. We test 1- to 10-Hop sketch graphs for a 1000-node fragment.

After finding optimum parameters for each case, we use those parameters for the comprehensive performance testing.

4.4.1 Disk-based SP Algorithm

The main purpose of the test in this section is to find the optimum fragment size for the best performance when using the disk-based SP algorithm suggested in [7]. In addition, for the optimum cache size of a distance matrix, we test 5 different parameters.

The results shown in this section are obtained by executing the disk-based SP algorithm, modified and having eliminated some parts. In Section 2.3, we described the disk-based SP algorithm and there are two places where the algorithm merges two fragments: source and destination fragments in finding a skeleton path, and two

consecutive fragments in which the algorithm fills out the skeleton path of two boundary nodes. After we eliminate those merging operations in the algorithm, we can speed up the execution of the disk-based SP algorithm. In addition to the conditions above, it does not pre-process queries with *QueryGraph*.

4.4.1.1 The Effect of Fragment Size

The fragment size affects the performance because materialized data, such as a distance matrix, or the number of nodes in a fragment, are decided by the fragment size and directly involved in the algorithm. Therefore, it is important to find the optimum size for a partitioned graph.

Figure 4.2 shows the performance difference according to the different fragment sizes for the graph of Connecticut. For each query type, we process 100 queries, and the time shown in the graph is an average time such query. The cache sizes for the fragment DB and distance-matrix DB are set to the number of the entries in the fragment DB and distance-matrix DB, i.e., the I/O activity is not a factor in the result of the test. As shown in the figure, a 1000-node fragment generates the best performance in every range of queries.

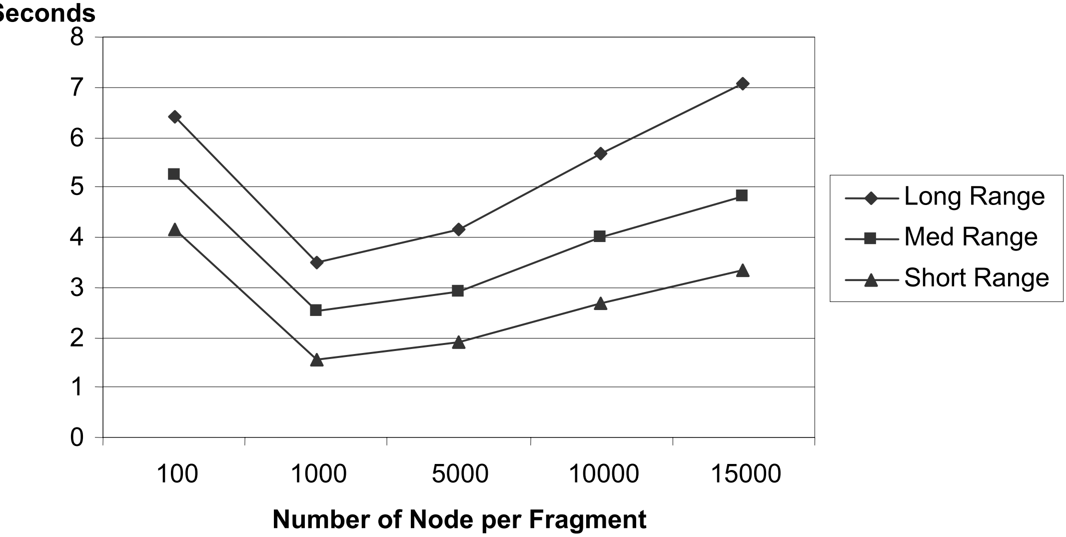


Figure 4.2 Calculation Time for the Different Size of Fragments

4.4.1.2 The Effect of the Cache Size of the Distance Matrix

The effect of the cache size for a distance matrix is tested in this section, and we report the result for a 1000-node fragment case since we determined that this size performs best on the Connecticut graph in Section 4.4.1.1. All the settings for the test are same as in Section 4.4.1.1 except the cache size of the distance-matrix DB. Figure 4.3 shows the result of the test cases, each of which ran through 100 queries and calculated an average per query. The 1000-node fragment of the Connecticut graph has a total of 347 distance matrices, and we test 5 different cache sizes: 10, 20, 30, 40, and 150. For each range of query sets, the result is very similar in that the best result is around 20 to 30. Even if we increase the cache size above 30, the calculation time changes little. From these results, we can ascertain that the cache size of around 10% of the total distance matrices in the distance-matrix DB is enough.

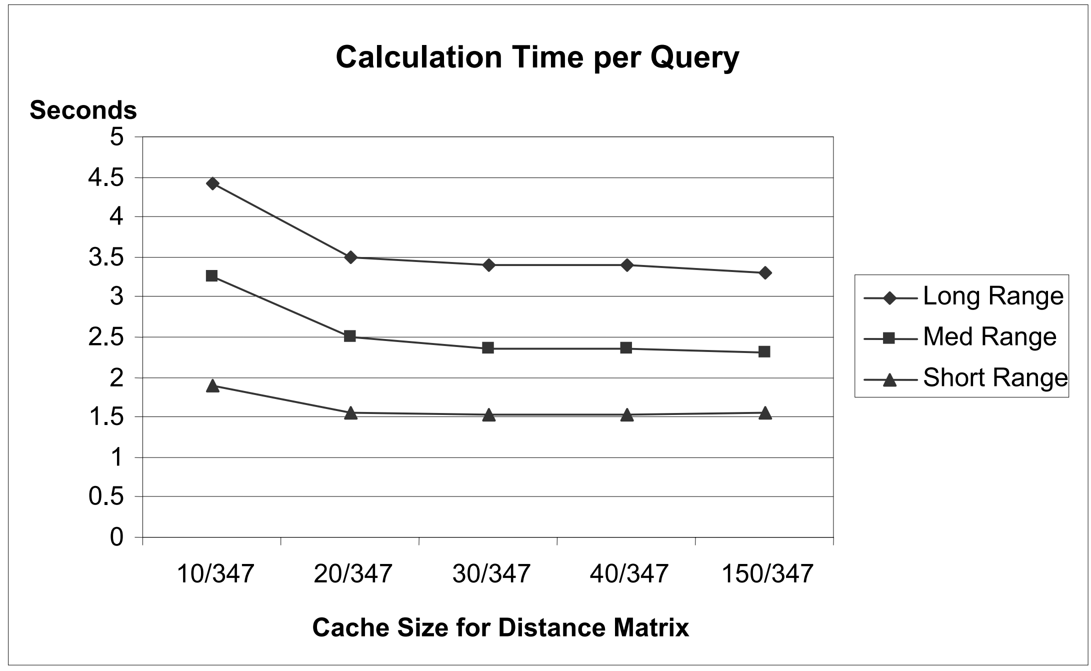


Figure 4.3 Calculation Time According to Different Cache Sizes

In short, the fragment of 1,000 nodes with the cache size of 30 for a distance matrix performs best in the Connecticut graph. Therefore, we primarily investigate the

case of a 1000-node fragment in the following sections. In the coming sections, all the tests are executed with 10% of the whole cache size for a distance matrix.

4.4.2 Pruning Algorithm Using *BSDistMatrix*

The advantage we expect from the pruning algorithms is to eliminate a number of boundary sets in a sketch graph so that the disk-based SP algorithm [7] uses less search space. In the disk-based SP algorithm, there are numerous I/O activities and calculations when the algorithm processes boundary nodes, which makes pruning search spaces important.

For the tests in this section, we do not use *QueryGraph* nor the query-grouping scheme, introduced in the batch disk-based SP algorithm. We test 300 queries, 100 queries for each query type. We use the modified disk-based SP algorithm described in Section 4.4.1, with the cache sizes of 2 for the fragment DB and 10% of the total distance matrices for the distance-matrix DB. We also modify the pruning algorithm for reason of efficiency. The pruning algorithm explained in chapter 3 first builds two SP trees rooted by the source and destination in their fragments respectively, and then calculates approximations for pruning. However, the disk-based SP algorithm also builds the SP tree from the source until it finds the destination. Therefore, we have the SP tree from the source in the source fragment during the disk-based SP algorithm. That makes it redundant that the pruning algorithm builds the SP tree from the source before executing the disk-based SP algorithm, so we use the SP tree generated by the disk-based SP algorithm and trigger the pruning algorithm as the completion of the SP tree. Finally, we need to build the SP tree from the destination only, in order to start the pruning algorithm. The trade-off is that the pruning algorithm may prune some of the boundary sets in the source fragment. With the modified pruning algorithm, we trigger the pruning algorithm after closing all the boundary nodes in the source fragment, so we do not take any advantage from those pruned boundary sets in the source fragment. Because of the trade-off, the number of boundary nodes accessed during the execution of the disk-based SP algorithm with x-Hop sketch graph is, in fact, less than the one with *BSDistMatrix* for

some cases, which cannot happen if we use the original scheme for the pruning algorithm with *BSDistMatrix* (Figure 4.6 in Section 4.4.3).

The time for the pruning process using *BSDistMatrix* is expected to be minimal since there is no complicated calculation involved. The only time-consuming task is applying Dijkstra's algorithm on the source and destination fragments in order to build an SP tree. Table 4.4 shows the average time per query for the pruning algorithm using *BSDistMatrix*, including loading it into the memory, calculating approximations, and pruning boundary sets, according to the different sizes of the fragments. The number of entries indicates the number of entries in the matrix of each case, which is the result of ($n \times n$), where n is the number of nodes (boundary sets) in a sketch graph.

	100 nodes	1000 nodes	5000 nodes	10000 nodes	15000 nodes
Time (Sec.)	0.14	0.055	0.19	0.29	0.37
No of Entries	11764900	120409	4356	841	441

Table 4.4 AverageTime per Query for Pruning Using *BSDistMatrix*

As in the table, the optimum fragment size for *BSDistMatrix* is interestingly a 1000-node fragment as well; the same optimum size as resulted in Section 4.4.1. The 100-node fragment takes more time in calculating approximations than the 1000-node fragment, even though building an SP tree in the 100-node fragment for the source and destination fragments takes less time than in the 1000-node fragment. This is the outcome because there are many more entries in *BSDistMatrix*, which means it also has more boundary sets to process. A fragment size of more than 5000 takes more than a 1000-node fragment because the time for building an SP tree in the destination fragment increases significantly as the size of a fragment grows.

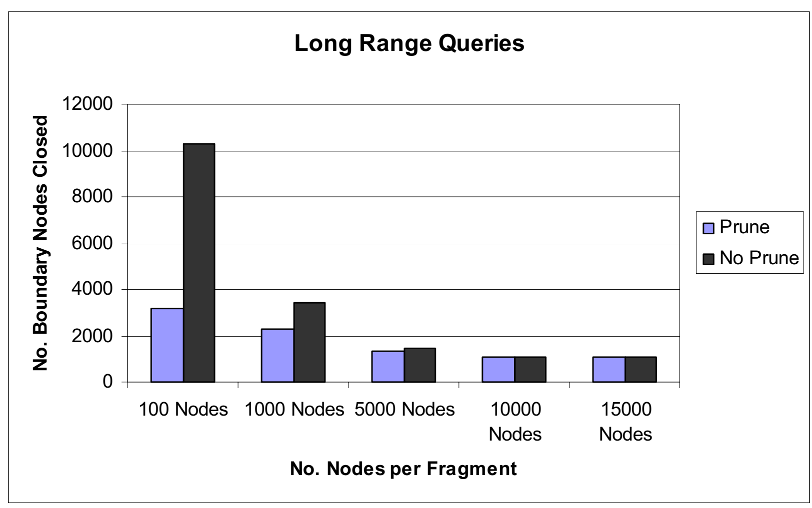
One of metrics for measuring the effectiveness of the pruning algorithm is to count the number of boundary nodes used during the calculation of skeleton paths. If a node in a sketch graph is pruned, then the disk-based SP algorithm does not include the boundary set which the node represents. Since a boundary set contains a number of boundary nodes, the more nodes (boundary sets) the pruning algorithm eliminates in the

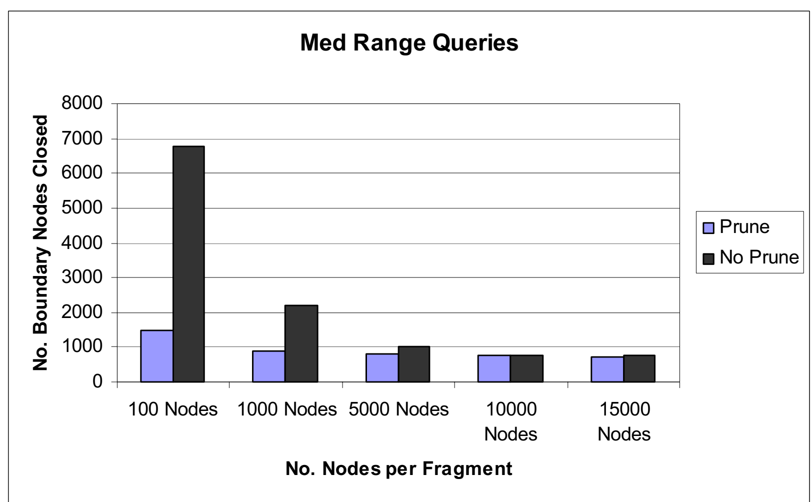
sketch graph, the less boundary nodes the disk-based SP uses during the calculation. Figure 4.4 shows how many boundary nodes we can save from the pruning algorithm. Each bar in the figure represents the average number of closed boundary nodes per query during the disk-based SP algorithm. For example, the algorithm needs to close about 2100 boundary nodes to calculate a skeleton path of a medium query without pruning in the 1000-node-fragment case. On the other hand, the algorithm needs just over 1000 boundary nodes with pruning, saving over 40% of accesses for boundary nodes. In closing a boundary node, the algorithm has to open and update the distances of neighbor boundary nodes, and the number of the neighbor boundary nodes is huge. In the case of the 1000-node fragment, each fragment has more than 100 boundary nodes, which means every boundary node has about 100 neighbor nodes. Therefore, in order to close one boundary node, the algorithm has to access 100 neighbor boundary nodes.

The figure also shows that the pruning algorithm does not work well with larger fragments. The reason is that we have a smaller number of boundary sets as we increase the size of each fragment. The difference between the approximations and the actual shortest distance becomes larger as the size of each fragment increases, so we lose the accuracy of approximations in larger fragments.

Figure 4.5 illustrates the average calculation time per query with and without the pruning algorithm. The queries are the same query set used in Section 4.4.1. For the pruning algorithm using *BSDistMatrix* to be effective, we should use fragment DB, in which each fragment has fewer than 5000 nodes.

The case of a 100-node fragment improves the most, but the performance is a little slower than that of the 1000-node fragment. Therefore, we can conclude that fragments with 1000 nodes are the best choice out of the 5 suggested fragment sizes for pruning algorithms. The other query sets with different distance ranges behave in a similar way, and the case of a 1000-node fragment works best.





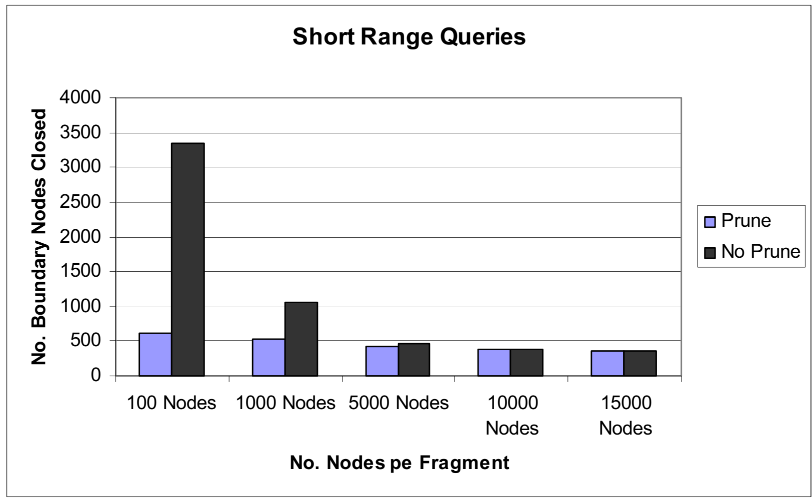
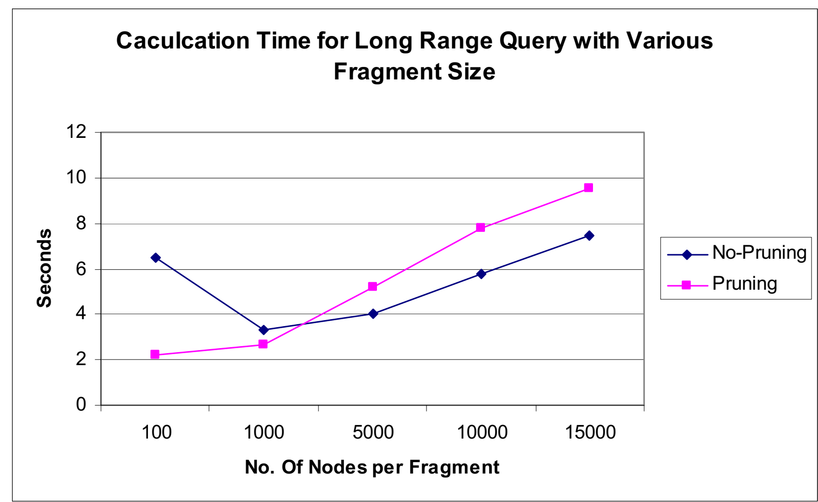
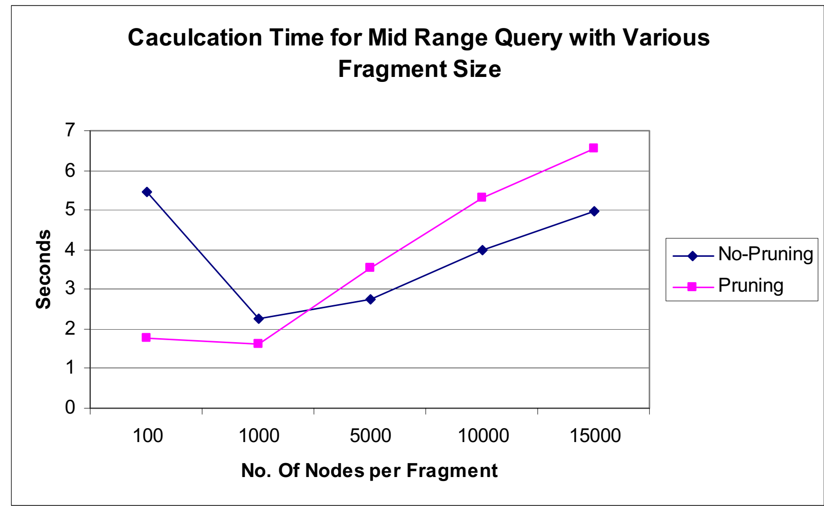


Figure 4.4 Average Number of Boundary Nodes Closed





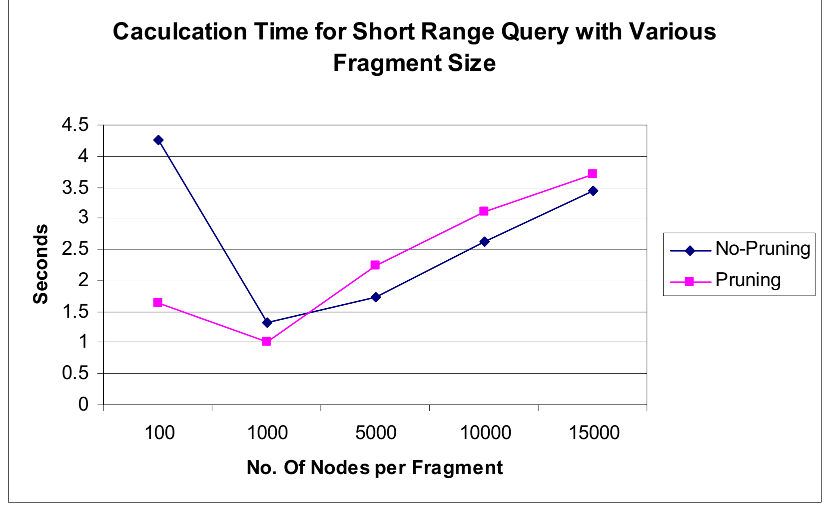


Figure 4.5 Average Calculation Time per Query

4.4.3 Pruning Algorithm Using an x-Hop Sketch Graph

Different from *BSDistMatrix*, a pruning algorithm with an x-Hop sketch graph requires more calculations: building the shortest-path trees from the source and destination in the source and destination fragments and finding the approximations using Dijkstra's algorithm on an x-Hop sketch graph. Therefore, the pruning procedure takes longer.

We show a 1000-node-fragment case for this test only, because we already know the 1000-node fragment works better than other cases and pruning with an x-Hop sketch graph is not different.

The testing environment is set as in Section 4.4.2 since we wish to compare the results. Therefore, except for the pruning part which is independent of the disk-based SP algorithm, we execute the disk-based SP algorithm with the same environment in Section 4.4.2.

First, we show a table of the augmented x-Hop sketch graphs according to x. Table 4.5 shows the number of edges in different augmented x-Hop sketch graphs and the time to do the whole procedure of pruning. The number of edges is important because the more edges a graph has, the more time it takes to calculate a shortest path using Dijkstra's SP algorithm. The number of edges peaks at $x = 6$ as x grows, and the average calculation time keeps increasing up to $x = 6$ and flatters after that. The reason is that one of the procedures for the pruning algorithm is making the augmented x-Hop sketch graph, which takes some time. The query set for the test is 100-long range queries with 1,000-node fragments. The other ranges of queries do not make any significant difference in the tests because all the tests use a very similar size of the augmented x-Hop sketch graph. Table 4.6 shows the average calculation time to obtain a skeleton path for a query according to x in the x-Hop sketch graph. For a long-range query, the disk-based SP algorithm works best when $x = 10$. For a medium-range query and a short-range query, the best choices are $x = 7$ and $x = 3$ respectively. Even though the best performances occur at $x = 10$, 7, and 3 for the long-, med- and short-range queries respectively, we choose $x = 5$, 4, and 3 respectively, in order to compare with the pruning algorithm using *BSDistMatrix*.

In order to compare the approximations between the x-Hop sketch graph and *BSDistMatrix*, we calculate the α- and β-approximations of all the boundary set pairs in the sketch graph with 1,000-node fragments. There are 347 boundary sets in the sketch graph, so we have 347^2 cases of the approximations, and then we categorize each case by the length of the approximations. For the x-Hop sketch graph, we only test the cases of x, where x = 1, 3, 5, 7 and 9. Table 4.7 shows the comparison for the β-approximations between the x-Hop sketch graph and *BSDistMatrix*. The figures in the table show the average ratio of the β-approximations with the x-Hop sketch graph to the β-approximations with *BSDistMatrix*. It clearly shows that the approximations with the x-Hop sketch graph become closer to the approximations with *BSDistMatrix* as x increases. Even in the case of x = 3 for a long-range query, the average approximation with the x-Hop sketch graph is only 7.3% longer than the one with *BSDistMatrix*. Table 4.6 shows the comparison for the α-approximations. The figures in the table also represents the average ratio of the α-approximations with the x-Hop sketch graph to the α-approximations with *BSDistMatrix*. Unlike the β-approximations, the changes become more radical as x increases. In the case of x = 3 for a medium-range query, the average α-approximation with the x-Hop sketch graph is only about 74% of the one with *BSDistMatrix*. If we compare the difference of the ratio between the α- and β-approximations in the same case, we can easily find that the pruning algorithm with x-Hop sketch graph calculates the β-approximations better than does the α-approximations.

To see how the pruning algorithm with an x-Hop sketch graph works according to x, we ran through 10 different x-Hop sketch graphs. Figure 4.6 shows the average number of closed boundary nodes during the calculations of skeleton paths using the disk-based SP algorithm, the same metric we used in Section 4.4.2 to test *BSDistMatrix*. In the figure, all three ranges of queries are tested, and we can observe how the pruning algorithm works with different x in x-Hop sketch graphs. For all three cases, the number of boundary nodes closed decreases as the number of hops in the x-Hop sketch graph increases. For the medium- and short-range queries, the number does not decrease dramatically beyond some points, and that can be interpreted to mean we do not need more than a certain x-Hop graph to have the best results. Therefore, we should choose

different x-Hop sketch graphs according to the length of the queries. Compared to *BSDistMatrix* in terms of the number of nodes closed, the x-Hop-sketch-graph approach works fairly well if we choose x carefully. For $x = 5$ for long-range queries, the algorithm closes around 2,500 boundary nodes, which is about 20% more nodes closed compared to *BSDistMatrix*. The calculation time including pruning, finding skeleton path, and finding actual path for a query will be shown in Section 4.4.4.

	$x = 1$	$x = 2$	$x = 3$	$x = 4$	$x = 5$	$x = 6$	$x = 7$	$x = 8$	$x = 9$	x=10
No. of Edges	1533	4109	6008	7212	7770	7881	7553	6668	5676	4719
Time for Pruning	0.14	0.16	0.21	0.25	0.28	0.31	0.32	0.32	0.33	0.33

Table 4.5 Number of Edges in Augmented x-Hop Sketch Graphs and Time to Calculate

	$x = 1$	$x = 2$	$x = 3$	$x = 4$	$x = 5$	$x = 6$	$x = 7$	$x = 8$	$x = 9$	x=10
Long	2.15	2.09	1.96	1.86	1.74	1.69	1.65	1.60	1.52	1.50
Med	1.58	1.34	1.09	1.10	0.99	0.97	0.96	0.98	0.99	0.98
Short	0.91	0.66	0.56	0.57	0.60	0.64	0.66	0.69	0.70	0.70

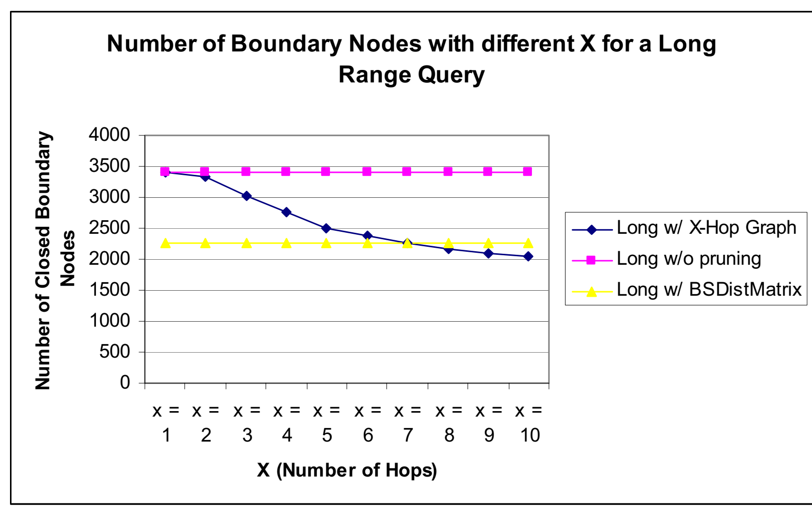
Table 4.6 Time to Calculate Skeleton Paths for Different Query Type

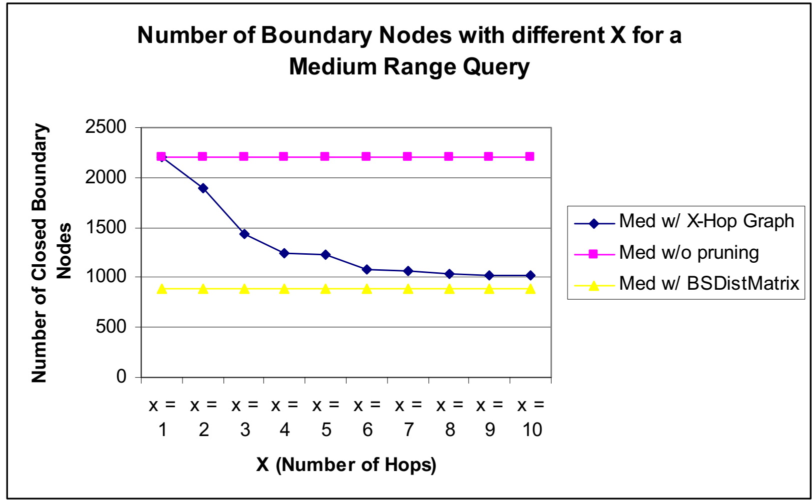
	$x = 1$	$x = 3$	$x = 5$	$x = 7$	$x = 9$
Long	1.36544	1.073148	1.034974	1.02139	1.012418
Med	1.349	1.062649	1.025602	1.006632	1.000628
Short	1.272677	1.027859	1.001732	1	1

Table 4.7 Comparison of the β-approximation

	$x = 1$	$x = 3$	$x = 5$	$x = 7$	$x = 9$
Long	0.239646	0.743585	0.872578	0.930323	0.952424
Med	0.265381	0.770313	0.907695	0.960071	0.99
Short	0.300416	0.862418	0.978902	0.999321	1

Table 4.8 Comparison of the α-approximation





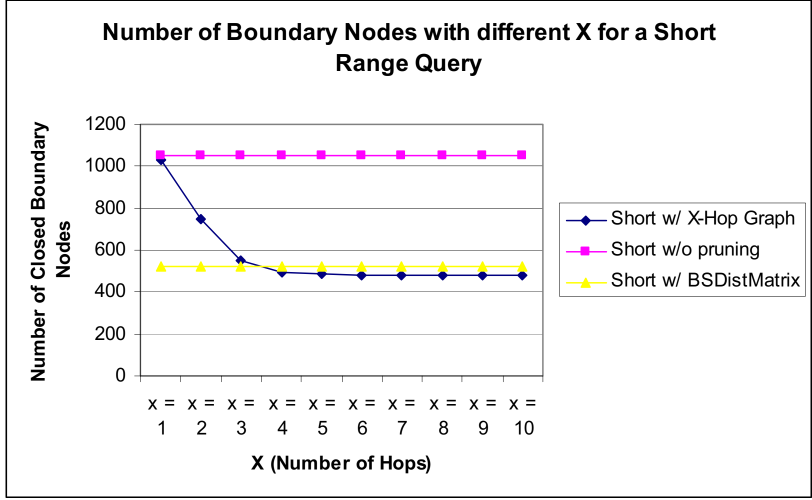


Figure 4.6 Average Number of Boundary Nodes Closed

4.4.4 Comprehensive Results

We have seen the results of individual algorithms so far, and, in this section, we examine results that give us the overall performance of all the algorithms in one combined algorithm. The description of algorithms tested for this section is as follows:

- Main memory version of Dijkstra's SP algorithm
  The algorithm takes the whole digital map into the main memory of the system and applies Dijkstra's SP algorithm to find the shortest paths. We assume that the digital map of Connecticut is loaded in the main memory before executing Dijkstra's SP algorithm.

- Disk-based SP algorithm
  With pre-computed data such as fragment DB and distance matrix DB, the disk-based SP algorithm calculates the shortest paths using the materialized data. The algorithm is explained well in [7], and, in fact, the program we use is the exact same algorithm with a little modification for efficiency. The modification we made for the algorithm is explained in Section 4.4.1.

- Disk-based SP algorithm with pruning using *BSDistMatrix*
  The disk-based SP algorithm except that it adopts the pruning algorithm using *BSDistMatrix* with the modification explained in Section 4.4.2.

- Disk-based SP algorithm with pruning using *BSDistMatrix* and grouping 10 queries
  In addition to the above, the algorithm groups 10 queries and processes them as described in Section 3.1.

- Disk-based SP algorithm with pruning using x-Hop sketch graphs
  The same disk-based SP algorithm except that it adopts pruning algorithm using x-Hop sketch graphs.

- Disk-based SP algorithm with pruning using *x*-Hop sketch graphs, and grouping 10 queries
  In addition to the above, the algorithm groups 10 queries and processes them as described in Section 3.1.

For the pruning algorithm using *x*-Hop sketch graphs, we take different *x*'s for different sets of queries: 3, 4, and 5 for short-, medium-, and long-range queries respectively. The queries are the same sets of queries used in the previous sections, and the fragment DB is a 1000-node fragment. The cache sizes for the fragment DB and the distance-matrix DB are 2 and 30 respectively. Additional to all of the above, all tests make use of *QueryGraph* before executing those algorithms, explained in Section 3.3.

To compare these algorithms, we investigate two metrics: the calculation time and I/O activity of a distance matrix. The calculation time is, of course, the most important metric since the whole point of the work being done is to reduce calculation time. The I/O activity of a distance matrix is also important because a distance matrix is a most-used data. For the I/O activity, we do not include the main memory version of Dijkstra's algorithm, because it does not have any I/O activities during calculation. Also, we do not include the algorithms with grouping queries in I/O-activity tests, because the grouping scheme affects only the filling out of real paths, with no relationship to finding skeleton paths.

Figure 4.7 shows the calculation time for different types of queries. It clearly shows that the main memory version of Dijkstra's SP algorithm performs the worst and that pruning algorithms in fact reduces the calculation time regardless of query types. The pruning algorithm using *BSDistMatrix* takes about 20% less calculation time for the long-range queries, compared to the original disk-based SP algorithm, and about 50% less compared to the main memory version of Dijkstra's SP algorithm. For the short–range queries, the advantage goes up to 30%. Combined with the grouping scheme, its performance for any type of queries is better than 30% of the original disk-based SP algorithm. The pruning algorithm with an *x*-Hop sketch graph also performs about 10% to 30% better than the disk-based algorithm according to the types of queries. With the

grouping scheme, the advantage goes up to 25% for the long-range queries, and 30% for the med- and short-range queries. If we compare two pruning algorithms, *BSDistMatrix* outperforms the *x*-Hop sketch graph by about 20%.

Figure 4.8 illustrates the performance in terms of accessing the distance matrix database during the process of finding skeleton paths in the algorithms. For this test, we use the cache size of 30 out of 347 total cache entries, about 10% of the total. As shown in the figure, using the pruning algorithm, we can reduce the I/O activity by more than 70% of the I/O activity of the original disk-based algorithm. The reason that the I/O activity of medium-range queries is slightly more than the one of long-range queries in the original disk-based algorithm is that the area covered by the medium-range queries in a graph is not much different from the one by the long-range queries since the disk-based algorithm itself is a greedy algorithm. Thus, we adopt the idea of pruning search spaces, and the results reveal that the pruning algorithm narrows search spaces. Less I/O activity for a distance matrix means the pruning algorithms make the disk-based algorithm access a lesser-distance matrix DB, which means fewer boundary-node accesses.

Overall, as proven by real-life test cases, the pruning algorithms combined with the grouping schemes reduce the calculation time as well as the amount of I/O activity.

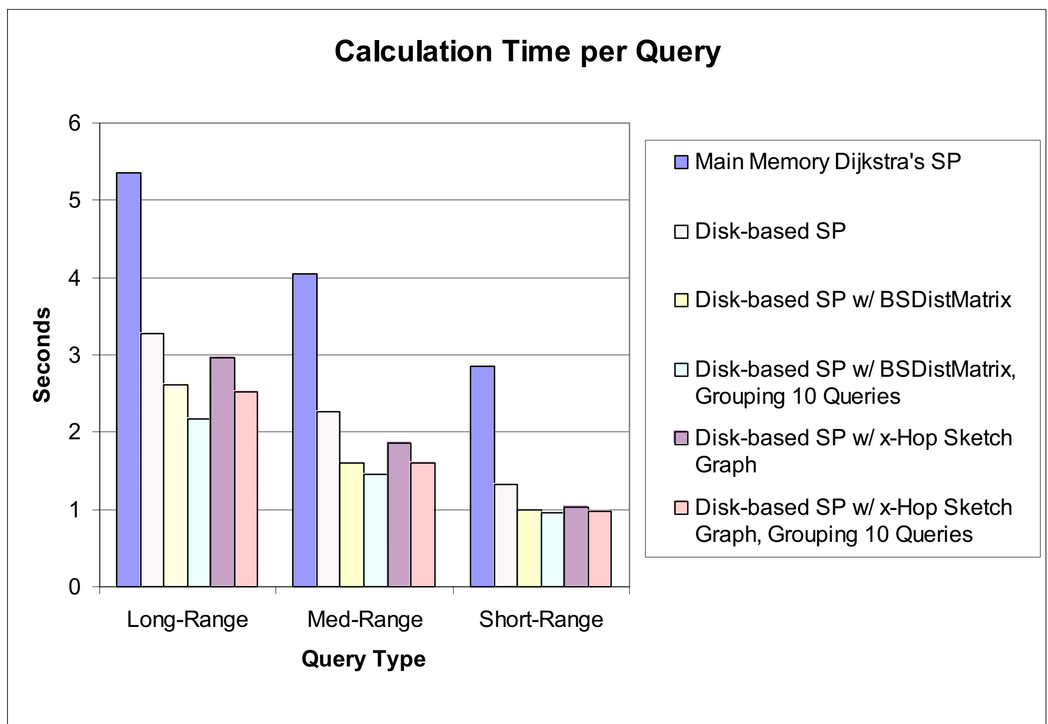


Figure 4.7 Calculation Time

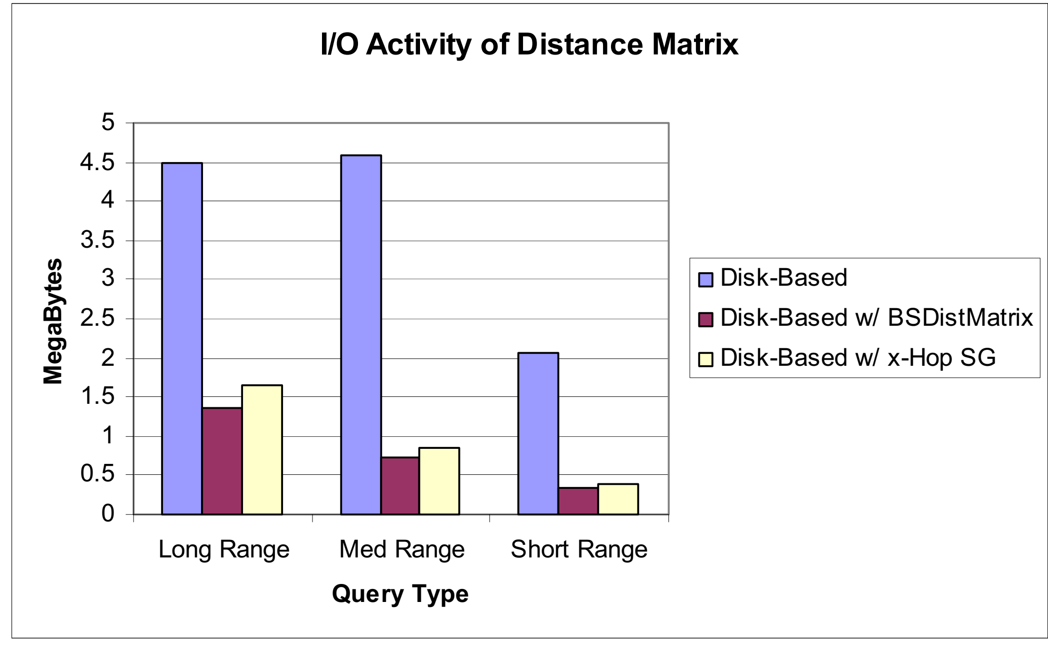


Figure 4.8 I/O Activity of Distance Matrix

Chapter 5
Conclusion and Future Research

5.1 Conclusion

We have studied various techniques for speeding up the Disk-based SP algorithm. We divided the Disk-based SP algorithm into three steps and implemented the algorithms to improve each step. The steps are query optimization, finding skeleton paths, and filling out the skeleton paths. For query optimization, we sorted the queries so that the step of filling out the skeleton paths accesses the fragment DB as little as possible. For finding skeleton paths, we suggested two pruning algorithms, each of which requires pre-computations to make the algorithms possible. Since the Disk-based algorithm uses the idea of Dijkstra's algorithm, it is basically a branch and bound algorithm. Our pruning algorithms narrow down the search spaces so that the Disk-based algorithm does not have to include unnecessary areas of a graph during calculation. For filling out the skeleton paths, we group a number of queries and process them together as in the previous step. When queries are grouped carefully, some of the queries may access common data, and our grouping technique helps the Disk-based algorithm to minimize the accessing of those common data.

The experimental results show that our algorithms improve the calculation time as well as I/O activities. In particular, both of the pruning algorithms significantly contribute to reducing the calculation time and I/O activities at the same time. Even if they need pre-computations, the benefits from the pruning algorithms make it worthwhile to do so. If we deal with a huge graph, such as a digital map of California, we should choose the

pruning algorithm using x-Hop sketch graphs because it takes less time to build such graphs. If a graph is small enough, we will probably choose the pruning algorithm using *BSDistMatrix* because it does not take too much time to build and the benefit can be maximized.

In conclusion, we improved the Disk-based SP algorithm by using various techniques while maintaining its essence—that it requires very little main memory.

5.2 Future Works

Future research includes topics such as enhancing multiple-query processing, reducing the building time of materialized data, and developing more efficient pruning algorithms.

In this thesis, we assume that every query comes into the system sequentially, which is very unlikely in real-life applications. For Disk-based algorithms to be useful, the algorithm must be able to process multiple queries at the same time.

Building *BSDistMatrix* and x-Hop sketch graphs constitutes a huge trade-off with the efficiency of the pruning algorithms. Even if we can control the calculation time of x-Hop sketch graphs by choosing x, it still takes a large amount of time if a graph is big. The other problem of *BSDistMatrix* and x-Hop sketch graphs is updating. If an original graph is updated, all, or part of the *BSDistMatrix* and x-Hop sketch graphs have to be updated as well, which requires a large amount of time. Therefore, minimizing the calculation time can solve the problem of building time as well as updating.

Our pruning algorithms work very well throughout all kinds of queries. However, there is still room for more pruning in a graph. The key to better pruning is calculating more accurate approximations. The approximation for the upper bound in our algorithm is close to the optimum value, but the one for the lower bound is not. Therefore, using different methodologies or different structures of *BSDistMatrix* could make approximations more accurate.

Bibliography

[1] Cherkassy B V, Goldberg A V and Radzik T., *Shortest Path Algorithms: Theory and Experimental Evaluation*, Mathematical Programming, Vol. 73, 129-174, June 1996.

[2] Greg Frederickson, Searching *Among Intervals and Compact Routing Tables*, Algorithmica, 448-466, 1996.

[3] Leonard Kleinrock, Farouk Kamoun, *Hierarchical Routing for Large Networks*, Computer Networks, 1:154-174, 1977.

[4] Ning Jing, Yun-Wu Huang, Elke Rudensteiner, *Hierarchical Optimization of Optimal Path Finding for Transportation Application*, In Proc. of ACM Conference on Information and Knowledge Management, 1996.

[5] Lars Arge, Gerth Stolting Brodal, Laura Toma, *On External-Memory MST, SSSP and Multi-way Planar Graph Separration*, Journal of Algorithms, November 2002.

[6] V. Kumar and E.Schwabe. *Improved Algorithms and Data Structures for Solving Graph problems in External Memory*, In Proc. IEEE Symp. On Parallel and Distributed Processing, pages 169-177, 1996.

[7] Edward P.F. Chan and Ning Zhang, *Finding Shortest Paths in Large Network Systems*, In Proceedings of 9th ACM International Symposium on Advances in GIS, November, 2001.

[8] Jesper L. Traff, *A Simple Parallel Algorithm for the Single-Source Shortest Path Problem on Planar Digraphs*, Journal of Parallel and Distributed Computing 60, 1103-1124 (2000).

[9] Apostolos N. Papadopoulos and Yannis Manolopoulos, *Multiple Range Query Optimization in Spatial Databases*, ADBIS 1998: 71-82, 1998.

[10] Sungwon Jung, and Sakti Pramanik, *An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps*, IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING, VOL. 14, NO. 5, 2002.

[11] Shashi Shekhar, Andrew Fetterer, and Brajesh Goyal, *Materialization Trade-Offs in Hierarchical Shortest Path Algorithms*, SSD 1997: 94-111, 1997.

[12] L. Arge, *The buffer tree: A new technique for optimal I/O-algorithms*. In Proc. Workshop on Algorithms and Data Structure, NLCS 955, pages 344-345, 1995.

[13] H.V. Jagadish, *Linear clustering of objects with multiple attributes*, Proceedings of the1990 ACM SIGMOD Conference, pp.332-342, Atlantic City, NJ, 1990.

[14] M. R. Garey and D. S. Johnson. *Computers and intractability: A guide to the theory of NP-completeness*. W.H. Freeman and Co, 1979.

[15] *Tiger/Line Files*, *1998*. Technical Documentation, US Department of Commerce Economics and Statistics Administraion, Bureau Of Census.

[16] 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.

[17] Ibrahim Kamel, Christos Faloutsos, *Hilbert R-tree: An Improved R-tree using Fractals*. VLDB 1994: 500-509

[18] E. W. Dijkstra, *A note on two problems in connection with graphs*, Numerische Mathematik 1 (1959), 269-271.

[19] Ning Zhang, *Shortest Path Queries in Very Large Spatial Databases*, a Thesis presented to the University of Waterloo in Computer Science, 2001.