Optimal Path Queries in Very Large Spatial Databases

by

Jie Zhang

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, 2005

Abstract

Researchers have been investigating the optimal route query problem for a long time. Optimal route queries are categorized as either unconstrained or constrained queries. Many main memory based algorithms have been developed to deal with the optimal route query problem. Among these, Dijkstra's shortest path algorithm is one of the most popular algorithms for the unconstrained route query problem. The constrained route query problem is more complicated than the unconstrained one, and some constrained route query problems such as the Traveling Salesman Problem and Hamiltonian Path Problem are NP-hard. There are many algorithms dealing with the constrained route query problem, but most of them only solve a specific case. In addition, all of them require that the entire graph resides in the main memory. Recently, due to the need of applications in very large graphs, such as the digital maps managed by Geographic Information Systems (GIS), several disk-based algorithms have been derived by using divide-and-conquer techniques to solve the shortest path problem in a very large graph. However, until now little research has been conducted on the disk-based constrained problem.

This thesis presents two algorithms: 1) a new disk-based shortest path algorithm (DiskSPNN), and 2) a new disk-based optimal path algorithm (DiskOP) that answers an optimal route query without passing a set of forbidden edges in a very large graph. Both algorithms fit within the same divide-and-conquer framework as the existing disk-based shortest path algorithms proposed by Ning Zhang and Heechul Lim. Several techniques, including query super graph, successor fragment and open boundary node pruning are proposed to improve the performance of the previous disk-based shortest path algorithms. Furthermore, these techniques are applied to the DiskOP algorithm with minor changes. The proposed DiskOP algorithm depends on the concept of collecting a set of boundary vertices and simultaneously relaxing their adjacent super edges. Even if the forbidden edges are distributed in all the fragments of a graph, the DiskOP algorithm requires little memory. Our experimental results indicate that the DiskSPNN algorithm performs better than the original ones with respect to the I/O cost as well as the running time, and the DiskOP algorithm successfully solves a specific constrained route query problem in a very large graph.

Acknowledgements

I would like to give great thanks to Professor Edward P.F. Chan, my supervisor, for his support and encouragement during my master's study. His valuable suggestions and comments lead directly to this thesis.

My special appreciation to Professor Peter Van Beek and Professor Ashraf Aboulnaga for taking their precious time reviewing this thesis as readers, and to Ms. Gloria Rose for proofreading this thesis.

I would also like to thank to the School of Computer Science in University of Waterloo and Bell Laboratory for funding this project. I am grateful for the helpful discussions with my friends Ning Zhang, Heechul Lim, Danny Zhang, and Jian Wang.

Most of all, I would like to thank my husband, my parents, and my brother. Their endless love and encouragement gave me confidence and enabled me to finish this thesis.

Contents

List of Tables

List of Figures

Abbreviations

BSDM – boundary set distance matrix

DM – distance matrix

DMDB – distance matrix database

SP – shortest path

SPT – shortest path tree

OP – optimal path

OPF – optimal path forest

OPT – optimal path tree

DiskSP – the disk-based shortest path algorithm in this thesis

DiskSPN – the improved disk-based shortest path algorithm in this thesis

DiskSPNN – the proposed new disk-based shortest path algorithm in this thesis

DiskOP - the proposed new disk-based optimal path algorithm in this thesis

NDM - a brute-force disk-based optimal path algorithm by re-computing distance matrices

NOPT - a brute-force disk-based optimal path algorithm by computing optimal path tress

WCSPP - Weight Constrained Shortest Path Problem

Chapter 1

Introduction

1.1 Optimal Route Query Problem in Spatial Databases

Optimal route queries can be divided into unconstrained and constrained route queries [23]. The most frequently asked queries, the shortest path (SP) queries belong to the former category, and some classical path problems such as the Traveling Salesman Problem and Hamiltonian Path Problem [6] fall into the latter category. In Geographical Information Systems (GIS), a user defined route query can involve multiple restrictions rather than just computing a path with the minimum cost. Constrained route queries have many applications in areas such as transportation, tourism, urban planning, and networks. Graphs are used to represent the road maps and the networks. In addition, we can label the vertices and the edges in a graph with a set of attributes like name and weight.

The unconstrained path problem which is to find an SP from the source to the destination in a graph has been well-studied. There are many mature SP algorithms such as Dijkstra's algorithm, Bellman-Ford algorithm and Floyd-Warshall algorithm [6]. However, when the main memory is not large enough to load the entire graph, these algorithms do not work properly. Recently, several disk-based algorithms ([19], [20], [21], [31]) adopt divide-and-conquer techniques to solve this particular problem. This thesis presents several techniques, including query super graph, successor fragment, sketch graph pruning with breadth first search, and open boundary vertex pruning to improve the performance of the algorithms proposed in [21] and [31]. Experimental results shows that the I/O cost of our new algorithm is only 40% of the algorithm in [31], and 65% of the algorithm in [21]. In addition, the search space of our algorithm is just one third of that of the algorithm in [31].

Many real-life constrained route optimization problems are difficult to address in practice. Even a simple constrained query as: "find an SP from A to B with a gas cost less than \$50", is NP-hard [14]. There are many papers ([1], [5], [9], [10], [15], [17], [25]) that deal with the constrained problem, but most solve only a specific case. Until now, little work has been conducted to find a constrained path in a very large graph. Another example of a frequently asked constrained query is "find an SP from A to B not via any area with bad weather conditions", which is a forbidden edge problem. Other real-world route queries such as "find an SP from A to B not via any express way", and "find an SP from A to B not via any traffic jam roads" are queries with forbidden edge constraint. The answer of these forbidden edge queries is an SP in the modified graph that is achieved by deleting all the edges in the affected area. When the graph is small enough to be loaded in the main memory, the problem could be solved easily. Otherwise, it is difficult. This thesis describes an efficient algorithm to find a shortest path not via any forbidden edge in a very large graph.

Chapter 2 discusses the survey of the related work. Chapter 3 and Chapter 4 propose a new disk-based SP algorithm and a new disk-based OP algorithm respectively. Chapter 5 shows the experimental results of the algorithms. Chapter 6 presents conclusions and future work.

1.2 Terminlogy

In order to clarify the description of the proposed algorithms in the thesis, all the frequently used terms are defined in this section. Since this thesis is based on the work of two previous disk-based SPs algorithms ([21], [31]), many terms have been defined in [21] and [31]. Here, those terms are briefly defined.

Definition 1. Directed Graph

The 3-tuple G=(V, E, w) is defined to be a *directed graph*, where V=$\{v_i \mid i \in [0, n-1]\}$

is the vertex set with size n, E=$\{e_k \mid e_k$ with properties $(v_i,\ v_j),\ i,j \in [0, n-1], k \in [0, m-1]\}$ is the directed edge set with size m, and w is a one-to-one function from the set of edges to non-negative numbers, denoted as w: $E \rightarrow w_k \in \Re^{\geq 0}$. Each directed edge e_k has three properties: a *head* vertex v_i, a *tail* vertex v_j, and a non-negative real number w_k representing its weight, which are denoted as *head*(e_k), *tail*(e_k) and $w(e_k)$ respectively. In addition, these three properties uniquely determine an edge in G. Also, multiple edges in a digraph are allowed. For example, Figure 1.1 illustrates a typical directed graph. Here, the vertices are labelled from 0 to 42, and (v_2, v_8, 6) denotes the edge from vertex 2 to vertex 8 with weight 6. Edge (v_7, v_9, 4) and (v_7, v_9, 5) are multiple edges from vertex 7 to vertex 9 with weights 4 and 5, respectively.

Definition 2. Digital Map

A *digital map* D=(V, E, w) is defined as a persistent graph in the secondary storage, where V, E, and w are the same as those defined in Definition 1. Since both a digital map and a graph are the same in theory and differ only in implementation, we use "graph" for describing the graph algorithms and other theoretical concepts.

Definition 3. Sub-graph

A *sub-graph* g=(V_g, E_g, w) of graph G=(V, E, w) has the following properties: $V_g \subseteq V$, $E_g \subseteq E$. Also, there exist two one-to-one functions f_v: $V_g \rightarrow V$, f_e: $E_g \rightarrow E$ such that $\forall v_i \in V_g$, $f_v(v_i)$=v_i, $\forall e_k \in E_g$, $f_e(e_k)$=e_k and $w(e_k)$= $w(f_e(e_k))$. Here, the edge weight function w is the same as that of G. According to the definition of a sub-graph, the vertices (edges) in the sub-graph are a subset of the vertices (edges) in the original graph.

Definition 4. Fragment

A *fragment* F=(V_F, E_F, w) is a connected sub-graph of G=(V, E, w). F and G use the same weight function w. The weight of an edge in a fragment is the weight of the corresponding edge in the original graph. A fragment is a special kind of sub-graph. There exists an edge connecting two vertices in a fragment, if and only if the two corresponding vertices in the original graph are adjacent. In Figure 1.1, the original graph is divided into six fragments as indicated by the dashed line. Each fragment is a sub-graph of the original

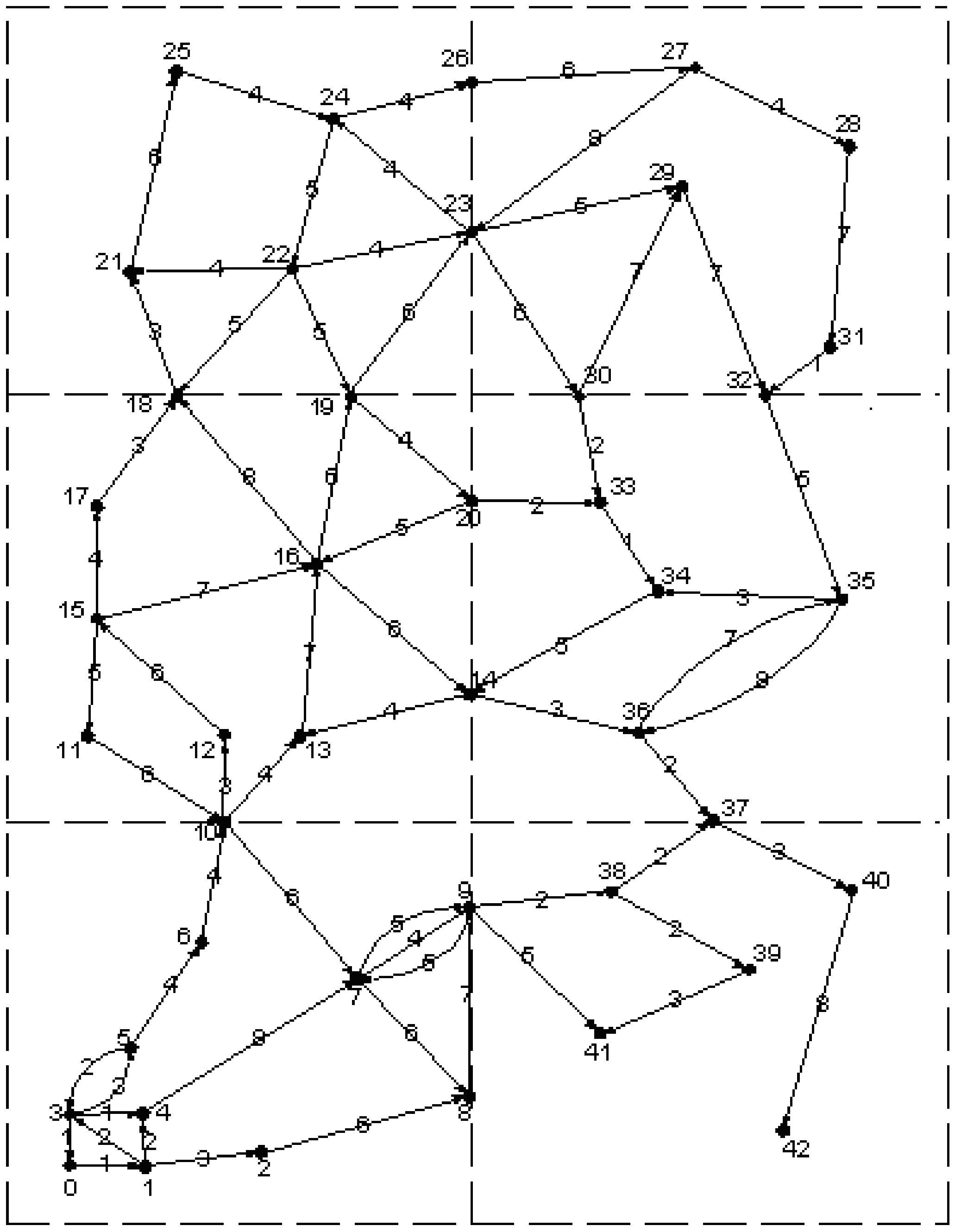


Figure 1.1: Sample directed graph

graph and when the fragments are merged, we obtain the original graph.

Definition 5. Partition, Fragment ID

A *partition* of graph $G(V,\ E,\ w)$ is a set of fragments $\{F_i{=}(V_i,\ E_i,\ w)\ \mid\ i{\in}[1,\ n],\ \cup V_i{=}V$, where i is the ID of $F_i\}$. It can be concluded that $\cup E_i{=}E$ and the previous fragments can be sorted according to their IDs.

Definition 6. Interior Vertex, Boundary Vertex

The vertices in fragment $F{=}(V_F,\ E_F,\ w)$ of graph $G(V,\ E,\ w)$ can be partitioned into two sets: V_i and V_b, where $V_F{=}\ V_b{\cup}V_i$. A vertex in fragment $v_i{\in}V_b$ *iff* there exists an adjacent vertex u of $f_v(v_i){\in}V$ such that $u{\notin}V_F$, implying that each boundary vertex connects to at least two fragments of its partition. The vertices in V_b are *boundary vertices*, and those in V_i are *interior vertices.* In Figure 1.1, vertex 8, 9, 10, 14, 18, 19, 20, 23, 26, 30, 32, and 37 are boundary vertices; the others are interior vertices.

Definition 7. Boundary Set

A *boundary set* is the set of all boundary vertices shared by two or more fragments, denoted as $BS[f_i,\ f_j,\ \ldots,\ f_k]$, where $f_i,\ f_j,\ \ldots,\ f_k$ are the fragments sharing the boundary vertices in the boundary set. The sorted $f_i,\ f_j,\ \ldots,\ f_k$ sequence uniquely determines the boundary set and is the ID of the boundary set.

A boundary vertex can be shared by multiple fragments, when its adjacent edges lie in these fragments. If a boundary vertex appears in more than one boundary set, the boundary sets are restricted to the boundary sets between two fragments. For example, for a boundary vertex v shared by fragments $f_1,\ f_2,\ f_3,\ v$ exists in boundary sets $[f_1,\ f_2]$, $[f_2,\ f_3]$, and $[f_1,\ f_3]$. To avoid too many boundary vertices in more than two boundary sets, and to make the process of finding boundary vertices shared by two fragments easier, all the boundary sets are between two fragments.

Definition 8. SD-value, OD-value, and Optimal Distance

The *SD-value* of vertex u to vertex v in graph G, denoted as $SD(u,\ v,\ G)$, is defined as the shortest distance from u to v in G.

The *OD-value* of vertex u to vertex v in graph G wrt forbidden edge set E_f , denoted as $OD(u,\ v,\ G,\ E_f)$, is the shortest distance from u to v in G not via any edge in E_f.

Definition 9. Optimal Route Query, Optimal Path, Optimal Distance, and Optimal Path Tree

In this thesis, we define an *optimal route query* $Q=(s,\ d,\ G,\ E_f)$ as a search to find a path p from vertex s to d with the minimum weight of all the paths from source vertex s to destination vertex d in graph G, not via any edge in forbidden edge set E_f. Path p is an *optimal path* (OP) of query Q with the following properties: for any edge $e \in p$, $e \notin E_f$, and $w(p)=SD(s,\ d,\ G')$, where $G'(V,\ E\text{-}E_f,\ w)$. Therefore, the OP can be thought of as an SP in graph G', which is a sub-graph of G by removing all the edges in E_f. The weight of path p is the *optimal distance* from s to d in G wrt E_f and denoted as $w(p)=OD(u,\ v,\ G,\ E_f)$.

A *shortest path tree* (SPT) T in $G'(V,\ E\text{-}E_f,\ w)$ is also an *optimal path tree* (OPT) in $G(V,\ E,\ w)$ wrt forbidden edge set E_f. Let s be the root of T and v be a tree node. Therefore, the path from s to v in T is an OP from s to v in G wrt forbidden edge set E_f.

Definition 10. Optimal Path Forest, Virtual Root, Ancestor, and Descendant

Given a vertex set $X=\{x_1,\ x_2,\ \ldots,\ x_n\}$ with a value set $C=\{c_1,\ c_2,\ \ldots,\ c_n\}$ in graph G and forbidden edge set E_f, where $c_i \in \Re^{\geq 0}$ is the attached value of x_i ($i \in [1,\ n]$), an *optimal path forest* (OPF) rooted at X in G is computed as follows:

- Add a vertex r into G with outgoing edges $e(r,\ x_i,\ c_i)$, where $1 \leq i \leq n$; here, name r as a *virtual root* and its adjacent edges as *virtual edges*;
- Compute an outgoing OPT T rooted at r in G wrt E_f;
- Remove virtual root r and its adjacent edges from T; the obtained data structure is an OPF, denoted as $OPF(X,\ C,\ G,\ E_f)$.

$OPF(X,\ C,\ G,\ E_f)$ consists of a set of sub-trees of T. The roots of these sub-trees are vertices in set X. For each vertex u in graph G, if there is a path from any vertex of X to

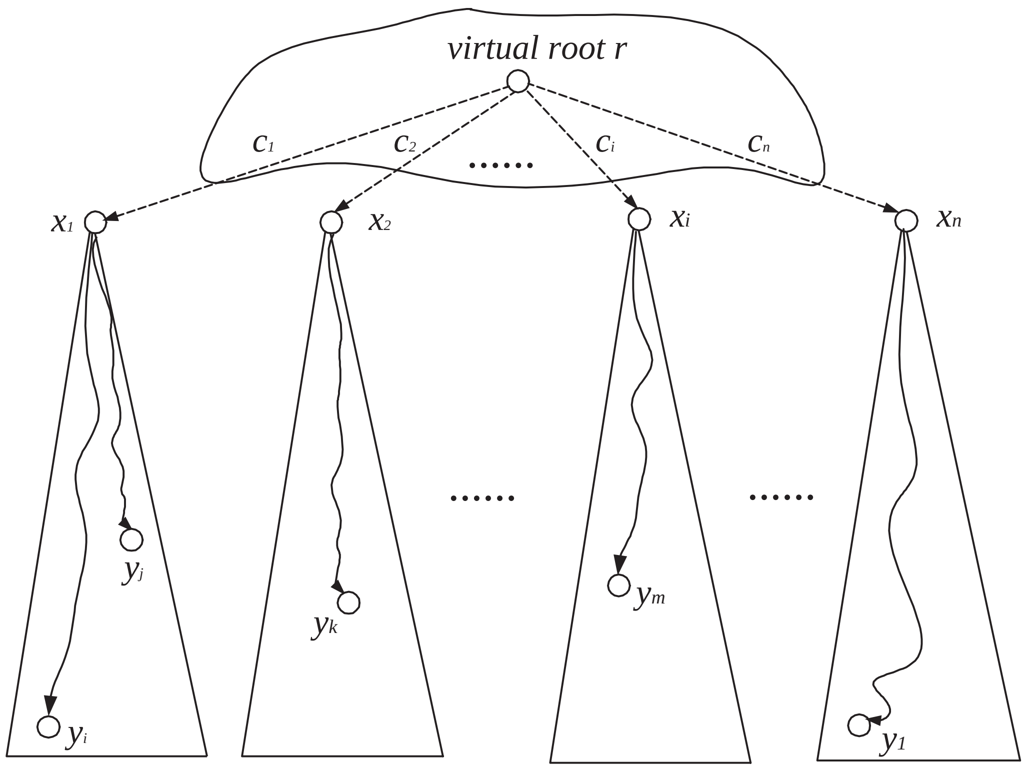


Figure 1.2: Optimal path forest rooted at X with value C

u, not via any forbidden edge of E_f, then u is a node of the forest. Let the sub-tree with root $x_i \in X$ $(1 \leq i \leq n)$ be the tree holding u. Then $c_i + OD(x_i,\ u,\ G,\ E_f) = min(\{c_j +\ OD(x_j,\ u,\ G,\ E_f\ )\ \mid\ 1 \leq j \leq n\})$, x_i is the *ancestor* of u, and u is the *descendant* of x_i. For example, in Figure 1.2, vertex y_k is the descendant of x_2 and x_2 is the ancestor of y_k.

Definition 11. Super Graph, Query Super Graph and Skeleton Path

A *super graph* $S=(V_S,\ E_S,\ w_S)$ of graph partition $\{F_1,\ F_2,\ \ldots,\ F_n\}$ has the following properties: $V_S = \{v_b\ \mid\ v_b$ is a boundary vertex in $F_k,\ k \in [1,n]\}$, $E_S = \{e(v_i,\ v_j,\ F_k)\ \mid\ v_i$ and v_j are the boundary vertices in fragment $F_k,\ k \in [1,\ n]\}$. For any $e_{ijk} =\ e(v_i,\ v_j,\ F_k) \in E_S$,

$w_S(e_{ijk})= SD(v_i,\ v_j,\ F_k)$, where v_i, v_j are the boundary vertices in fragment F_k, $k\in[1,\ n]$. The vertices of the super graph are the boundary vertices in the fragments. For each pair of the boundary vertices, there is an edge connecting them. Edge $e(v_i,\ v_j,\ F_k)\in E_S$ maps to a shortest path from v_i to v_j in fragment F_k. If there is no path from v_i to v_j in F_k, its weight is infinity. Figure 1.3 presents the super graph of the sample graph. For each pair $(v_i,\ v_j)$ of the boundary vertex inside the same fragment, there is an edge from v_i to v_j and another edge from v_j to v_i.

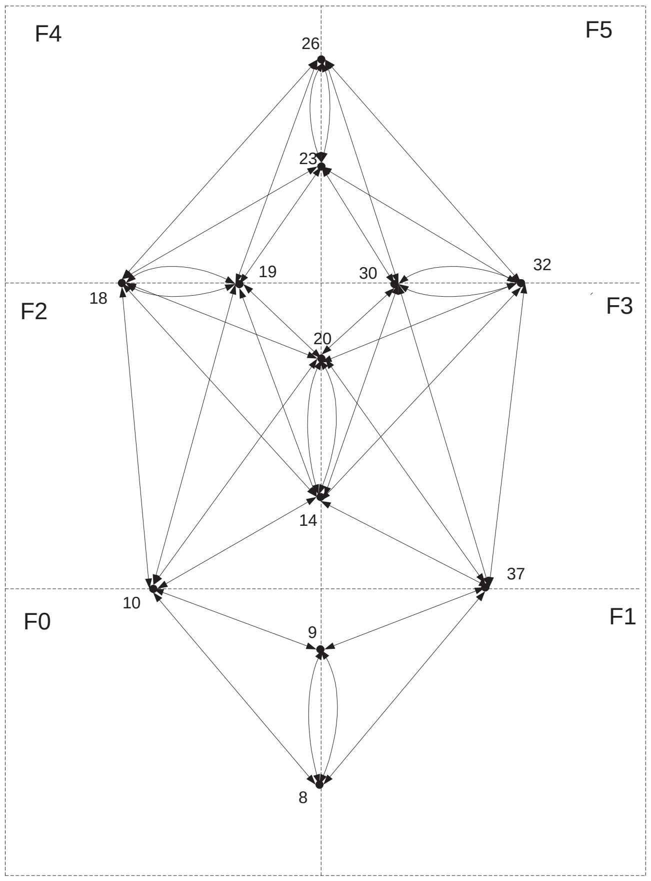


Figure 1.3: Super graph

For an optimal query Q=(s, d, G, E_f), a *query super graph* S_Q=(V_Q, E_Q, w_Q) corresponds to Q with the following properties: V_Q=$V_S\cup\{s,\ d\}$, E_Q= $E_S\cup E_{src}\cup E_{dst}$, where

E_{src}=$\{e(s, v_i, F_s)$ ∣ v_i is a boundary vertex in the fragment F_s holding s. If both s and d are in F_s, v_i is a boundary vertex in F_s or $d\}$, E_{dst}=$\{e(v_i, d, F_d)$ ∣ v_i is a boundary vertex in the fragment F_d holding $d\}$. For any e_{ijk}=$e(v_i, v_j, F_k)\in E_Q$, $w_Q(e_{ijk})$=$OD(v_i, v_j, F_k, E_f)$, where v_i and v_j are boundary vertices in fragment F_k, v_i, $v_j\in V_Q$, and $k\in[1, n]$. The vertices and edges in a super graph (query super graph) are named *super nodes* and *super edges* respectively.

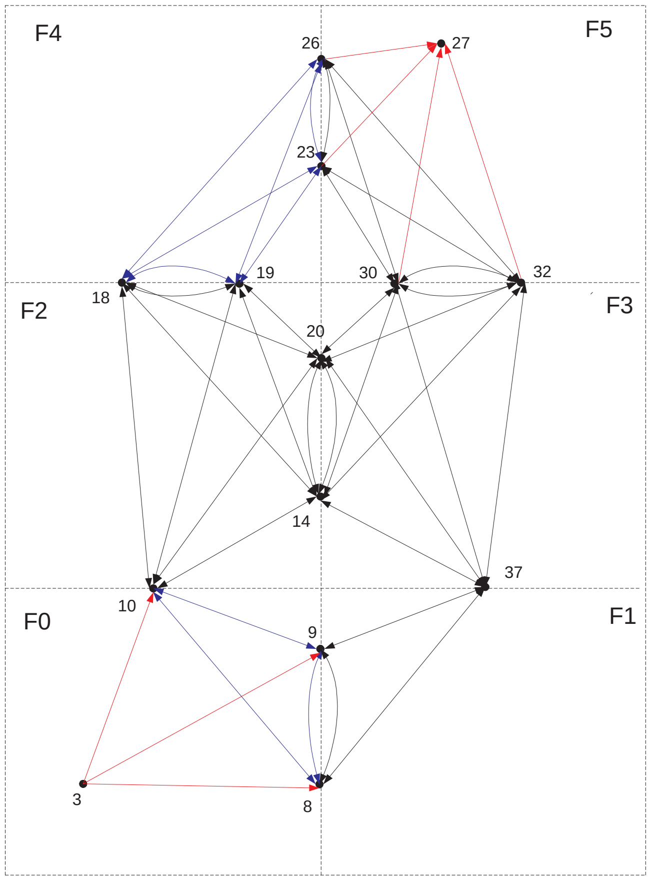


Figure 1.4: Query super graph

A query super graph is constructed according to a super graph by adding the source and the destination vertices and their adjacent edges. Due to the forbidden edge constraint, the weights of the super edges are the optimal distances inside the fragment from one vertex to the other vertex. If a fragment contains a forbidden edge, the fragment is *affected*; otherwise, it is *unaffected*. In Figure 1.4, the query super graph of Q=(v_3, v_{27}, G, E_f), where G is the sample graph in Figure 1.1 and E_f={(v_5, v_6, 4), (v_{24}, v_{22}, 5)}. F_0 and F_4 contain a forbidden edge and are affected fragments, whereas the remaining fragments are unaffected fragments. F_0 is the source fragment and F_5 is the destination fragment. The adjacent edges of v_3 are e(v_3, v_8, F_0), e(v_3, v_9, F_0), and e(v_3, v_{10}, F_0). The adjacent edges of v_{27} are e(v_{23}, v_{27}, F_5), e(v_{26}, v_{27}, F_5), e(v_{30}, v_{27}, F_5), and e(v_{32}, v_{27}, F_5).

A *skeleton path* is an SP in a super graph or a query super graph consisting of super nodes and super edges.

Definition 12. Optimal Skeleton Path Tree and Optimal Skeleton Path

A SPT T in a query super graph is named an *optimal skeleton path tree* in original graph G. T is an outgoing tree with root s, a tree node v is a super node and a tree edge e is a super edge in the query super graph. The path from s to tree node v in T is an optimal skeleton path in G.

Definition 13. Predecessor, Successor, and Predecessor (Successor) Fragment

Let skeleton path p_s be $\langle e_1, e_2, \ldots, e_m \rangle$, where e_i is a super edge in the query super graph, $i \in [2, m]$, $tail(e_i)$= $head(e_{i-1})$ and $head(e_i)$=$tail(e_{i+1})$. Let e_{i-1}= (v_x, v_y, F_k) and e_i=(v_y, v_z, $F_{k'}$). Therefore, there is a boundary vertex sequence $\langle v_x, v_y, v_z \rangle$ corresponding to edge sequence $\langle e_{i-1}, e_i \rangle$. The *predecessor* of boundary vertex v_y in path p_s is v_x and the *successor* of v_y is v_z. Obviously, fragments F_k and $F_{k'}$ are adjacent to each other. The fragments containing v_y can be denoted by F_k and $F_{k'}$, where F_k is the *predecessor fragment* of v_y and $F_{k'}$ is the *successor fragment* of v_y.

Definition 14. Closed, Open, Relaxed, Finished, and Un-finished

During the process of computing an optimal skeleton path starting from s in query super graph S_Q= (V_Q, E_Q, w_Q), when the distance from s to super node $u \in V_Q$ is finalized, that is $OD(s, u, G, E_f)$ is obtained, u is *closed*. Otherwise, it is *open*. The proposed disk-based optimal path algorithm (DiskOP) is a variant of Dijkstra's algorithm. The

DiskOP iteratively chooses a super node u with the minimum distance of all the open super nodes, adds u into closed vertex set, and relaxes all the outgoing super edges of u. When the DiskOP finishes the relaxation of all the super edges leaving vertex u, u is *relaxed*. Otherwise, it is *un-relaxed*. When a super node is both closed and relaxed, it is *finished*. Otherwise, it is *un-finished*.

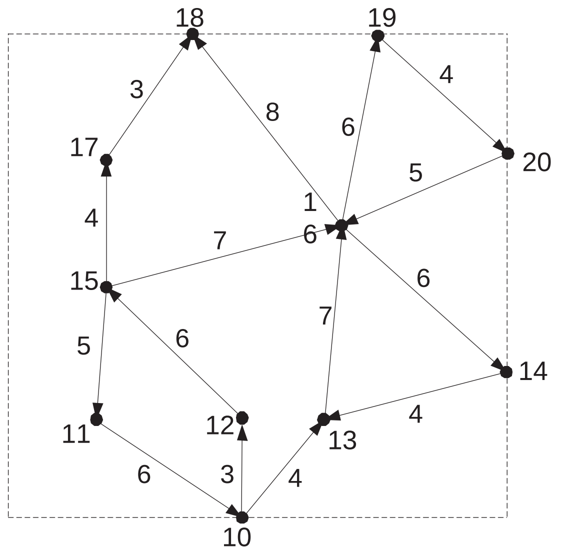
Definition 15. Pivot, Pivot Path, Pivot Fragment, Pivot Graph and PD-Value

For each boundary set BS_i of partition $\{F_1, F_2, \ldots, F_n\}$, we randomly choose a boundary vertex $v \in BS$ as the *pivot* of BS such that there exists a one to one mapping f_p: $BS \rightarrow V_p$, where BS is the set of boundary sets in the partition, and V_p is the set of pivots. The path between any two pivots inside a fragment is called a *pivot path*. In fragment F_i, we can construct one or more pivot paths for each pair of pivots and merge all these paths into a graph, which is a *pivot fragment*. *Pivot graph* G_p is obtained by merging a set of pivot fragments; that is $G_p = \cup F_{pi}(i \in [1,\ n])$. Figure 1.5(b) provides an example of the pivot fragment of F_2. Vertices v_{10}, v_{20}, and v_{19} are the pivots of $BS[F_0,\ F_2]$, $BS[F_2,\ F_3]$, and $BS[F_2,\ F_4]$, respectively. Path $p_{10-19} = \langle e(v_{10},\ v_{13},\ 4),\ e(v_{13},\ v_{16},\ 7),\ e(v_{16},\ v_{19},\ 6)\rangle$ is an SP from v_{10} to v_{19}. There is no path from v_{19} to v_{10}. $p_{10-20} = \langle e(v_{10},\ v_{13},\ 4),\ e(v_{13},\ v_{16},\ 7),\ e(v_{16},\ v_{19},\ 6),\ e(v_{19},\ v_{20},\ 4)\rangle$ is an SP from v_{10} to v_{20}. There is no path from v_{20} to v_{10}. $p_{19-20} = \langle e(v_{19},\ v_{20}\ ,\ 4)\rangle$ is an SP from v_{19} to v_{20}; $p_{20-19} = \langle e(v_{20},\ v_{16}\ ,\ 5),\ e(v_{16},\ v_{19}\ ,\ 6)\ \rangle$ is an SP from v_{20} to v_{19} . If only one path is materialized for each ordered pivot pair, then F_{p2} consists of the edges in paths p_{10-19}, p_{10-20}, p_{19-20} and p_{20-19}.

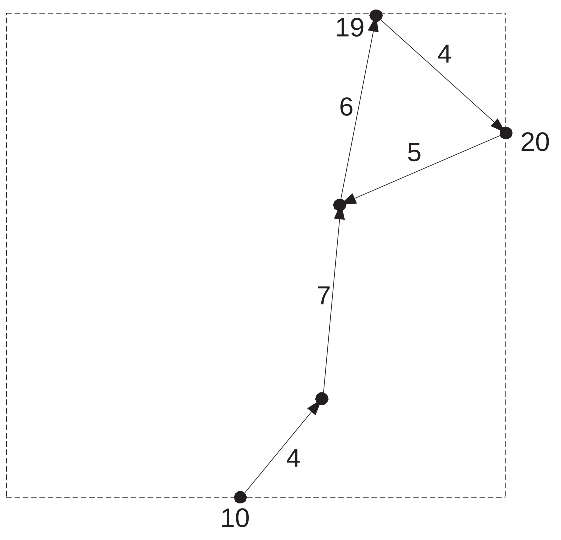
Let v_i and v_j be two pivots inside fragment F_k. *PD-value* from v_i to v_j in F_k is the optimal distance from v_i to v_j in F_{pk} wrt forbidden edge set E_f, where F_{pk} is the pivot fragment of F_k, $PD(v_i,\ v_j,\ F_k,\ E_f) = OD(v_i,\ v_j,\ F_{pk},\ E_f)$.

Definition 16. α-value, β-value

α-value of vertex set X to vertex set Y in graph G is the minimum value of the shortest distances from any vertex $x \in X$ to any vertex $y \in Y$, and $\alpha(X,\ Y,\ G) = min(\{SD(x,\ y,\ G)\ \mid\ x \in X,\ y \in Y\})$, where *min* is the minimum function. Also α-value is defined as the minimum value of the shortest distances between any vertex $x \in X$ and vertex v in graph



(a) Fragment F_2



(b) Pivot fragment F_{p2}

Figure 1.5: Pivot fragment

G. Therefore, $\alpha(v,\ X,\ G)=\ min(\{SD(v,\ x,\ G)\ \mid\ x{\in}X,\ v$ is a vertex in $G\})$; and $\alpha(X,\ v,\ G){=}min(\{SD(x,\ v,\ G)\ \mid\ x{\in}X,\ v$ is a vertex in $G\})$.

Similarly, β-*value* is the maximum value of the shortest distances from any vertex $x{\in}X$ to any vertex $y{\in}Y$, and $\beta(X,\ Y,\ G){=}max(\{SD(x,\ y,\ G)\ \mid\ x{\in}X,\ y{\in}Y\})$, where max is the maximum function. β-value can be extended to a vertex and a vertex set, denoted as either $\beta(v,\ X,\ G)$ or $\beta(X,\ v,\ G)$, respectively, and defined as $\beta(\{v\},\ X,\ G)$ or $\beta(X,\ \{v\},\ G)$, respectively.

Definition 17. Sketch Graph and Query Sketch Graph

Sketch graph $K{=}(V_k,\ E_k,\ w_k)$ of graph partition $\{F_1,\ F_2,\ \ldots,\ F_n\}$ has the following properties:

- $V_k{=}\{v_i\ \mid\ v_i$ corresponds to some boundary set B_i in fragment F_p, where $p{\in}[1,\ n]\}$;
- $E_k{=}\{e(v_i,\ v_j,\ F_p)\ \mid\ v_i$ and v_j correspond to some boundary sets B_i and B_j in F_p respectively, where $p{\in}[1,\ n]\}$;

- for any $e_{ijp}=e(v_i, v_j, F_p)\in E_k$, $w_k(e_{ijp})=SD(pv_i, pv_j, F_p)$, where pv_i, pv_j are the pivots of v_i and v_j, respectively, and both of them are in fragment F_p ($p\in[1, n]$).

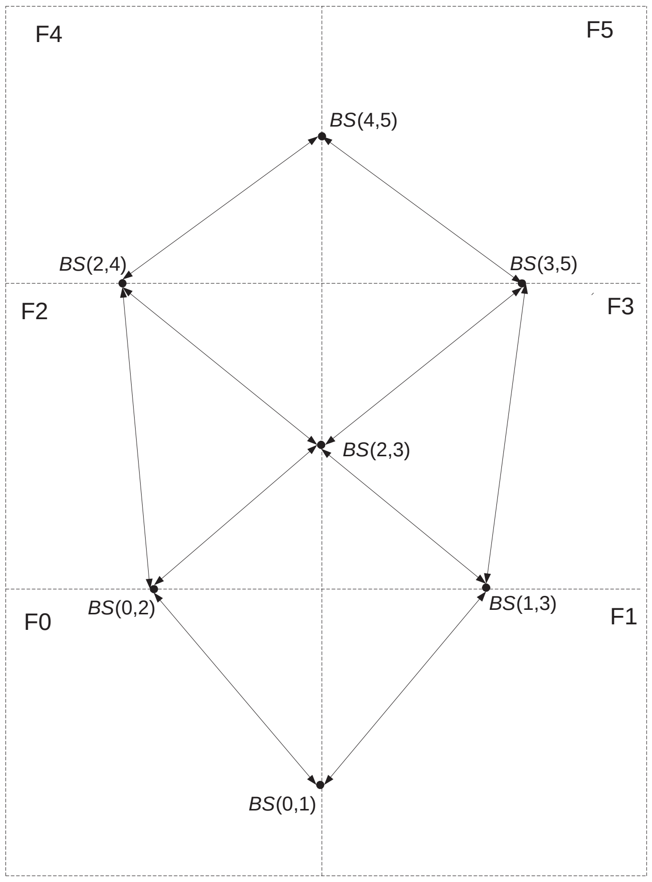


Figure 1.6: Sketch graph

In a sketch graph, each boundary set has an attribute to indicate its pivot. Figure 1.6 gives the sketch graph of the example super graph in Figure 1.3. The vertices and edges in a sketch graph are called *sketch nodes* and *sketch edges*, respectively. Since the boundary vertices of a boundary set are shared by the same set of fragments, a boundary set can be contracted into a sketch node to easily indicated the adjacency of the boundary sets. For example, to find the adjacent super edges of a boundary vertex v in a super graph with

$BS[F_1, F_2]$ as the boundary set holding v, we can retrieve the adjacent sketch node set $BSET$ of $BS[F_1, F_2]$ efficiently due to the small size of sketch graph. Therefore, $BSET \cup \{\mathrm{BS}[F_1, F_2]\}$ contains all the boundary sets of fragment F_1 and F_2, and all boundary vertices belonging to the boundary set in $BSET \cup \mathrm{BS}[F_1, F_2]$ excluding v are the adjacent super nodes of v in a super graph.

Given an optimal query Q=(s, d, G, E_f), a *query sketch graph* K_Q= (V_{KQ}, E_{KQ}, w_{KQ}) corresponds to Q with the following properties:

- $V_{KQ}=V_K \cup \{s, d\}$, $E_{KQ}= E_K \cup E_{Ks} \cup E_{Kd}$, where $E_{Ks}=\{e(s, v_i, F_s) \mid v_i$ corresponds to some boundary set B_i in fragment F_s holding s, or $v_i= d$ if $d \in F_s\}$, $E_{Kd}=\{e(v_i, d, F_d) \mid v_i$ corresponds to some boundary set B_i in fragment F_d holding $d\}$;

- For any $e_{ijp}=e(v_i, v_j, F_p) \in E_k$, $w_{KQ}(e_{ijp})=PD(pv_i, pv_j, F_p, E_f)$, where pv_i, pv_j are two pivots in fragment F_p ($p \in [1, n]$), and are the pivots of v_i and v_j, respectively;

- For any $e_{si}=e(s, v_i, F_s) \in E_{Ks}$, $w_{KQ}(e_{si})=OD(s, pv_i, F_s, E_f)$, where pv_i is the pivot of v_i in fragment F_s holding s;

- For any $e_{id}=e(v_i, d, F_d) \in E_{Kd}$, $w_{KQ}(e_{id})=OD(pv_i, d, F_d, E_f)$, where pv_i is the pivot of v_i in fragment F_d holding d.

The query sketch graph is constructed according the sketch graph by adding source and destination vertices and their adjacent edges, which are from the source to the pivots in the source fragment and from the pivots to the destination in the destination fragment, respectively. When a fragment is unaffected, the weights of all the sketch edges inside the fragment remain the same as those in the sketch graph. Otherwise, we must compute the weights of the sketch edges in the pivot fragment of an affected fragment. Figure 1.7 illustrates the query sketch graph of the sample query super graph in Figure 1.4. Since F_0 and F_4 are affected, the weights of the sketch edges inside them need to be updated.

Definition 18. Distance Matrix Database

Distance matrix (DM) is a data structure that contains the SP distances between any ordered pair of boundary vertices in the same fragment. Let m be the number of boundary

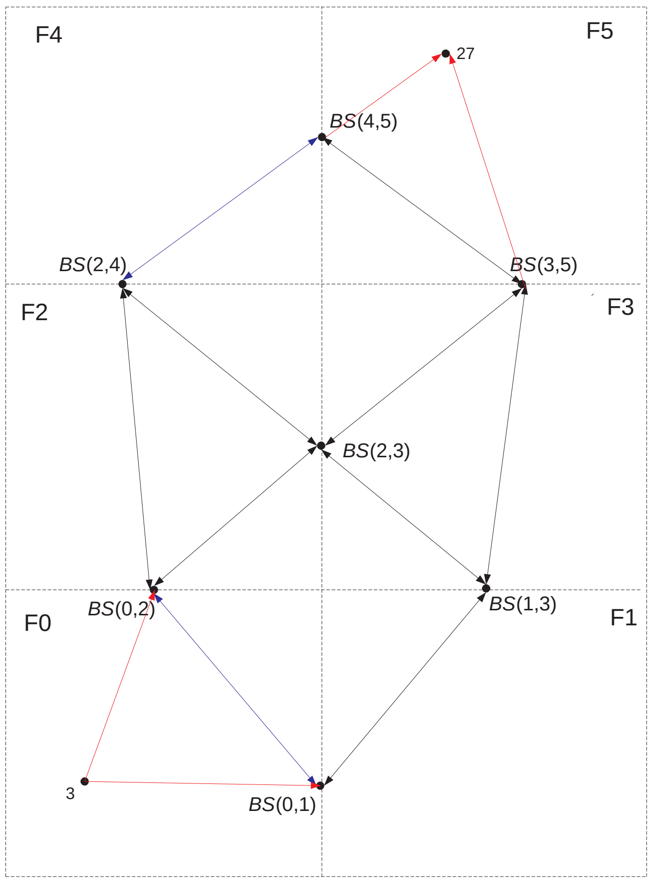


Figure 1.7: Query Sketch graph

vertices in fragment F_k. The matrix contains $(m \times m)$ entries. $DM_k = \{SD(v_i, v_j, F_k) \mid v_i, v_j$ are boundary vertices in $F_k\}$.

Distance matrix database (DMDB) consists of the distance matrices of all the fragments in a graph. In other words, DMDB contains the weights of all the super edges in a super graph. When the super graph is too large to be loaded into the main memory, the DMDB and the sketch graph are used to compute an SP in a graph.

Definition 19. Boundary Set Distance Matrix

Boundary set distance matrix (BSDM) is a data structure that contains the α-value and the β-value between each pair of boundary sets in the sketch graph. Let m be the number of boundary sets, BSDM has $(m \times m)$ entries and is $\{(\alpha(v_i,\ v_j),\ \beta(v_i,\ v_j))\ \mid\ v_i,\ v_j \in V_K\}$. The α-value (β-value) of the boundary set pair helps us to estimate a lower (upper) bound of an SP distance efficiently. Since the optimal distance is no greater than the shortest distance, the lower bound is still valid for optimal route queries. However, the lower bound to an optimal query is not as tight as that to a shortest path query with the same source and destination.

Definition 20. α-Approximation and β-Approximation

α-approximation from vertex u to vertex v in graph G, denoted as $\alpha A(u,\ v,\ G)$, is defined to be a value no greater than the shortest distance from u to v in G, that is $\alpha A(u,\ v,\ G) \leq SD(u,\ v,\ G)$. Therefore, α-approximation is the lower bound of the shortest distance from u to v in G. The α-approximation can be generalized to denote the lower bound of the minimum shortest distances between two vertex sets, or a vertex and a vertex set. Let X be a vertex set. $\alpha A(u,\ X,\ G)$ is less than or equal to the minimum value of the shortest distances from u to all the vertices of X; that is, $\alpha A(u,\ X,\ G) \leq min(\{SD(u,\ v,\ G)\ \mid\ v \in X\})$. Then, $\alpha A(X,\ u,\ G) \leq min(\{SD(X,\ u,\ G)\ \mid\ v \in X\})$, and $\alpha A(X,\ Y,\ G) \leq min(\{SD(u,\ v,\ G)\ \mid\ u \in X,\ v \in Y\})$, where $X,\ Y$ are vertex sets, and $u,\ v$ are vertices.

Similarly, *β-approximation* is defined to be an upper bound value of the shortest distance in graph G with the following properties: $\beta A(u,\ v,\ G) \geq SD(u,\ v,\ G)$; $\beta A(u,\ X,\ G) \geq min(\{SD(u,\ v,\ G)\ \mid\ v \in X\})$; $\beta A(X,\ u,\ G) \geq min(\{SD(X,\ u,\ G)\ \mid\ v \in X\})$; and $\beta A(X,\ Y,\ G) \geq min(\{SD(u,\ v,\ G)\ \mid\ u \in X,\ v \in Y\})$, where $X,\ Y$ are the vertex sets, and $u,\ v$ are the vertices.

Definition 21.γ-Approximation

γ-approximation from vertex u to vertex v in graph G wrt forbidden edge set E_f, denoted as $\gamma A(u,\ v,\ G,\ E_f)$, is defined to be a value no less than the optimal distance from u to v in G wrf E_f, that is, $\gamma A(u,\ v,\ G,\ E_f) \geq OD(u,\ v,\ G,\ E_f)$. Clearly, γ-approximation is the upper bound of the corresponding optimal distance.

Chapter 2

Survey of Related Work

In this chapter, the literature for both constrained and unconstrained route optimization problems is reviewed. The approaches to deal with these two problems are introduced in Sections 2.1 and 2.2, respectively.

2.1 Constrained Route Optimization

A constrained optimal route query is to find an OP, satisfying a set of constraints from the source to the destination in a given graph. Of all the paths satisfying the user-defined constraints, an OP is a path with the minimum weight. The constraints can be on the vertices or the edges, or on both of them. A constrained query can involve more than one weight metric, and aggregate functions. Many constraints are based on some aggregate functions (*e.g*, find an SP from town A to town B with a toll fee less than 100 dollars), which is a *Weight Constrained Shortest Path Problem* (WCSPP), defined as: attach a cost and a weight (or a vector of weights) to each edge in the graph, and then find a path with the minimum cost from the source to the destination such that the path's total weight is less than a specified value (or a vector of values). WCSPP is NP-hard, as proved in [14]. In the worst case, the running time is exponential wrt the number of the vertices in the given graph. Section 2.1.1 reviews several K shortest paths algorithms to solve the constrained route optimization problem by enumeration. Section 2.1.2 introduces approaches to solve the WCSSP. For the constraints such as avoiding certain areas or roads, the problem becomes easier. In Section 2.1.3, the dynamic SP algorithms are introduced to avoid the re-computation from scratch.

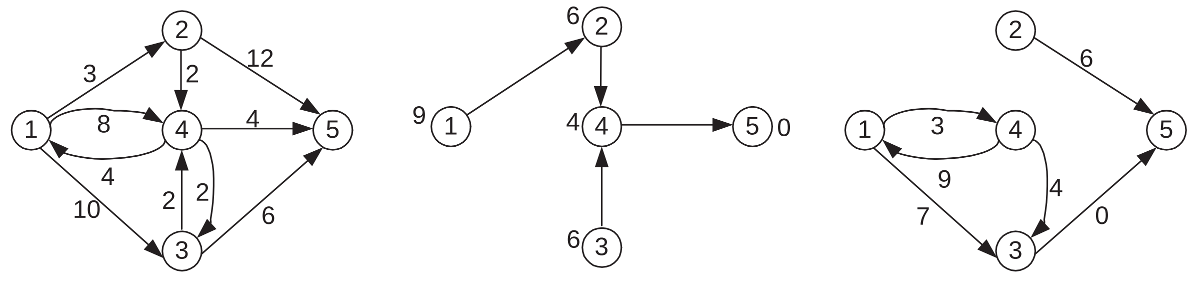
2.1.1 *K* Shortest Paths Algorithms

Enumeration is an alterative to obtain an OP for a constrained route query Q. The idea is simple, which is to list the first, second, ..., K^{th} SP from the source to the destination in the given graph until the one satisfying all the restrictions. Then, the K^{th} SP is an OP to Q. The algorithms, described by D.R. Shier [29], David Eppstein [11], E. Martins et al. [24], and Victor M.Jimenez et al. [18], find the K SPs which may not be simple. Yen [30], E. Martins et al [22], and John Hershberger[16] propose solutions for the K simple SPs.

K Shortest Non-simple Paths Algorithms

In this section, three K shortest non-simple paths algorithms are described briefly. Note that the returned paths may contain loops.

For source s and destination d in a digraph G, Eppstein's algorithm (Epp) [11] generates an implicit representation of those paths from s to d. The K SPs can be found in order of increasing weight with time $O(m+nlogn+KlogK)$. The Epp algorithm is the asymptotically fastest known one solving the problem. After finding a SPT T rooted at d, the Epp can compute *sidetrack value* for each edge $e(u,\ v)$, which is $sc(e)=SD(v,\ d,\ G)\text{-}SD(u,\ d,\ G)+w(e)$. The sidetrack value of an edge denotes the increased weight by going through this edge instead of taking the SP to d. Then, a tree of paths is built, where each tree node is represented by a set of edges corresponding to a path from s to d, and the node's weight is the sum of the sidetrack value of the path edges. Any child node in the tree has one more edge than its parent node, indicating that the weight of the path to which the parent node corresponds is no more than that of the child node. Since any path in the given graph consists of a set of SPT edges and a set of non SPT edges, a node in the tree of paths can be represented as an ordered non-tree edge sequence. Figure 2.1(d) is the tree of paths with source v_1, destination v_5 for the graph G in Figure 2.1(a). The Epp first computes a SPT T rooted v_5 in Figure 2.1(b). For each edge of T, its sidetrack value is zero while for a non-tree edge, the value is greater than or equal to zero (Figure 2.1(c)). The Epp builds the tree of paths TP from an SP from v_1 to v_5, which is the root of TP denoted as "{}". The children of the root are {3}, {4}, {6}, {7}, and {9}, which are the non-tree edges labelled by sidetrack value in Figure 2.1(c), because the tails of these non-tree edges are on the SP from v_1 to v_5. Node {3} corresponds to path $\langle v_1, v_4, v_5\rangle$ and the children of node {3} are {3, 4}, {3, 9}, representing path $\langle v_1, v_4, v_3, v_4, v_5\rangle$ and $\langle v_1, v_4, v_1, v_2, v_4, v_5\rangle$, respectively, because the tails of the edge labelled by 4 and 9 are on the SP from the head of edge labelled by 3 to the destination. For example, {3, 7} is not the child of {3} since there is no path v_1 to v_5 consisting of exactly the two non SPT edges $e(v_1, v_4)$, $e(v_1, v_3)$ and other SPT edges. In this way, the Epp builds the tree of paths and easily enumerates the K shortest paths from the source to the destination.


(a) Digraph G (b) Incoming SPT T rooted at v_5 (*e.g.,SD*(v_1, v_5, G)=9) (c) Edges in G-T_5 labelled by sidetrack value

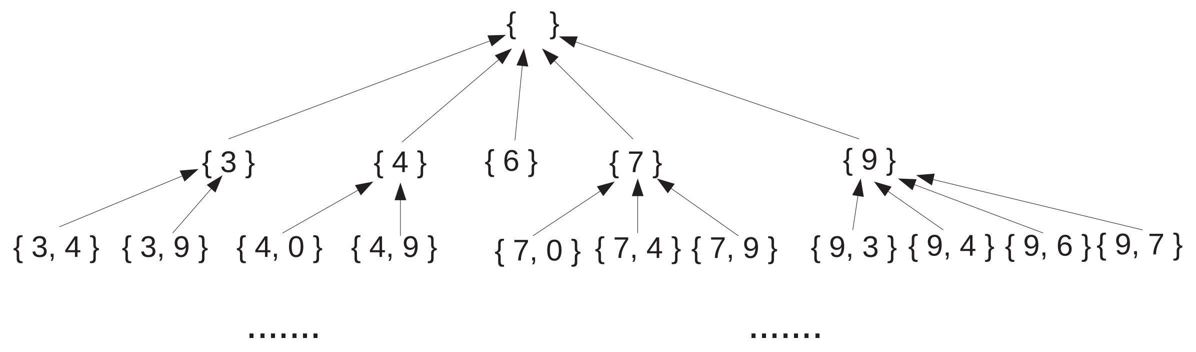

(d) Tree of paths TP

Figure 2.1: Example of the Epp algorithm

Victor M. Jimenez et al. presented *Recursive Enumeration Algorithm* (REA) [18] to recursively compute each new path by visiting, at most, the vertices in the previously computed path from s to d. The REA attaches a heap of candidate paths to each vertex, from which we can choose the next SP from source node s to the vertex. The REA computes the k^{th} shortest path from s to v by choosing the path with the minimum weight from v's candidate path set. Let p^k(v) denote the k^{th} SP from s to v. The most important task in the algorithm is to maintain v's candidate path set by replacing the latest found SP $p^k(v)$ $=p^{k'}(u)\diamond e(u,\ v)$ with $p^{k'+1}(u)\diamond e(u,\ v)$, where u is the predecessor vertex of v on the path $p^k(v)$, and $\diamond$ is used to concatenate two paths. Thus, the path with the minimum weight in the candidate path set is the $(k+1)^{st}$ SP from s to v. The REA finds the K SPs in time $O(m{+}Knlog(m/n))$.

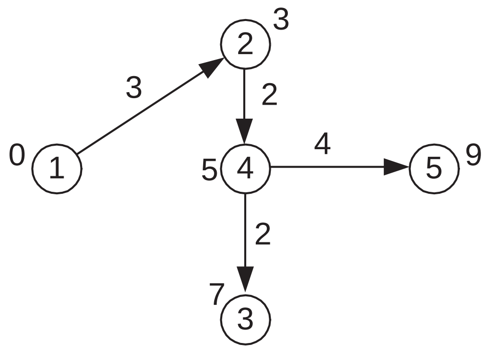


Figure 2.2: Outgoing SPT T rooted at v_1

Now, an example to find the first two SPs from v_1 to v_5 in graph G of Figure 2.1(a) with the REA is described. Denote $C^k(v)$ as the candidate path set for the k^{th} SP. Figure 2.2 gives an outgoing SPT rooted at v_1 in G. Because $C^1(v_5)$=$\{p^1(v_2)\diamond e(v_2,\ v_5)$, $p^1(v_3)\diamond e(v_3,\ v_5)$, $p^1(v_4)\diamond e(v_4,\ v_5)\}$, and $p^1(v_5)$=$p^1(v_4)\diamond e(v_4,\ v_5)$, $C^2(v_5)$=$\{p^1(v_2)\diamond e(v_2,\ v_5)$, $p^1(v_3)\diamond e(v_3,\ v_5)$, $p^2(v_4)\diamond e(v_4,\ v_5)\}$. Then $p^2(v_4)$ is computed as follows: since $C^1(v_4)$ =$\{p^1(v_1)\diamond e(v_1,\ v_4)$, $p^1(v_2)\diamond e(v_2,\ v_4)$, $p^1(v_3)\diamond e(v_3,\ v_5)\}$ and $p^1(v_4)$=$p^1(v_2)\diamond e(v_2,\ v_4)$, $C^2(v_4)$= $\{p^1(v_1)\diamond e(v_1,\ v_4)$, $p^2(v_2)\diamond e(v_2,\ v_4)$, $p^1(v_3)\diamond e(v_3,\ v_5)\}$. Now compute $p^2(v_2)$ is computed, because $C^1(v_2)$=$\{p^1(v_1)\diamond e(v_1,\ v_2)\}$ and $p^1(v_2)$=$p^1(v_1)\diamond e(v_1,\ v_2)$, $C^2(v_2)$=$\{p^2(v_1)\diamond e(v_1,\ v_2)\}$.

Since v_1 is the source, $p^2(v_1)$=$p^1(v_4)\diamond e(v_4,\ v_1)$ and $w(p^2(v_1))$=9, $p^2(v_2)$=$p^2(v_1)\diamond e(v_1,\ v_2)$, $w(p^2(v_2))$=9+3=12. Hence, $p^2(v_4)$=$min(C^2(v_4))$=$p^1(v_1)\diamond e(v_1,\ v_4)$, and $w(p^2(v_4))$=8. As a result, $p^2(v_5)$=$min(C^2(v_5))$=$p^2(v_4)\diamond e(v_4,\ v_5)$=$\langle v_1,\ v_4,\ v_5\rangle$ and $w(p^2(v_5))$=12.

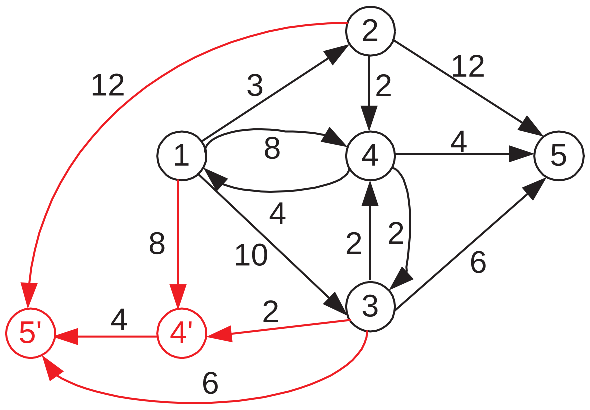


Figure 2.3: Resulting graph G_2

The idea of the MSA algorithm proposed by E. Q. V. Martins et al. in [24] is to construct a sequence of growing graphs G_1, G_2, ..., G_k such that the first SP in G_k is the k^{th} SP in G, where G=G_1. The MSA constructs G_k based on G_{k-1} by finding the first vertex on the $(k-1)^{st}$ SP with more than one incoming edge, builds G_k, and then determines the next SP in G_k. For example, the first SP from v_1 to v_5 in graph G of Figure 2.1(a) is $\langle v_1,\ v_2,\ v_4,\ v_5\rangle$, and v_4 is the first vertex on the path with more than one incoming edge. Vertex v_4' is created and new edges $e(v_1,\ v_4',\ 8)$, $e(v_3,\ v_4',\ 2)$ are added. Hence, the SP from v_1 to v_4' is $\langle v_1,\ v_4\rangle$. Then, a new vertex v_5' is created, which is the successor of v_4' on the first SP, and new edges $e(v_4',\ v_5',\ 4)$, $e(v_2,\ v_5',\ 12)$, $e(v_3,\ v_5',\ 6)$ are added. As a result, the SP from v_1 to v_5' is $\langle v_1,\ v_4',\ v_5'\rangle$, and the resulting graph is G_2 in Figure 2.3. The second SP is $\langle v_1,\ v_4,\ v_5\rangle$. This idea is straightforward and easily understood. But when the graphs are constructed, the number of vertices and edges grows quickly. The authors present a technique to avoid the increase on the number of edges, thus reducing the space complexity. After the SP from s to each vertex is computed, the total time required by the MSA to output K SPs in the order of increasing weight is $O(Km)$ in the worst case.

Among these algorithms, the Epp is outstanding due to its low asymptotic complexity. However, the Epp includes an initial stage to build an implicit representation graph from which K SPs are then computed quickly. If K is not large enough, the time to construct the implicit graph is not worth to the effort. Under certain conditions, the MSA and the REA run faster than the Epp, according to the experimental results in [24] and [18]. The K SPs computed by these algorithms may contain some loops. After all loops in the paths are removed, the number of distinct paths is probably far less than K. In a real-world optimal path problem, the optimal paths are most likely simple. As a result, the algorithms for finding the K shortest simple paths are favored.

K Shortest Simple Paths Algorithms

Since there is a simplicity restriction on the K SPs problem, intuitively, it is more difficult to solve than the one without any restriction. There are two principal approaches dealing with the K SPs problem: *label setting* and *deviation*. Label setting (e.g., the REA) is based on an Optimality Principle which asserts that there is an SP formed by the shortest sub-paths [23]. Because the K shortest simple paths problem does not satisfy this principle, we can no longer use it. The main idea of a deviation algorithm (e.g., the Epp and the MSA) is to construct a tree of paths containing K SPs. The theory to justify deviation algorithms is still valid when the goal is to find the K shortest simple paths between a pair of vertices. Therefore, the deviation algorithms are our only choice for the problem.

Yen's algorithm [30] is classical for finding the K shortest simple paths between a source node and a destination node in a given graph with non-negative loops. It can be applied in both directed and undirected graphs. The deviation idea is derived from this algorithm which constructs the k^{th} SP based on the k-1 SPs that have already been computed. The k^{th} SP consists of two sub-paths: a path from the source to some deviation vertex i which is a sub-path of the $(k-1)^{st}$ path, and the SP from vertex i to the destination. To satisfy the simplicity constraint, the latter sub-path does not include any vertex in the former one, and the successor of i in the k^{th} SP differs from any of i's successors among the k-1 available SPs with the same sub-path from the source to vertex i of the $(k-1)^{st}$ path. In this algorithm, list A is used to store the k-1 shortest simple paths that have been found, and list B is used to store the candidate paths of the k^{th} shortest simple path. After the termination of the algorithm, the K shortest simple paths between the two ordered given nodes are in list A. For example, the first SP from v_1 to v_5 in graph G (Figure 2.1(a)) is $\langle v_1, v_2, v_4, v_5\rangle$; based on this, the second shortest simple path can be computed. The deviation path on v_1 is $\langle v_1, v_4, v_5\rangle$, that on v_2 is $\langle v_1, v_2, v_5\rangle$, and that on v_4 is $\langle v_1, v_2, v_4, v_3, v_5\rangle$. Hence, list A contains the first SP and list B holds the previous three deviation paths. The one with the minimum weight should be extracted from B and add into A so that the path is the second shortest simple path, which is $\langle v_1, v_4, v_5\rangle$. The time complexity of this algorithm is $O(Kn^3)$. Compared with those K SPs algorithms mentioned in the previous section, the algorithm is rather slow due to the simplicity constraint.

The MPS algorithm [22] employs the idea of the sidetrack value to significantly reduce the number of arithmetic operations. The algorithm computes a set of deviation paths for each well-determined path. The merit of the algorithm is that it is unnecessary to remove the edges and the vertices from the given graph. The MPS concatenates two simple paths to be a deviation path, which is probably not simple. The first simple path is a sub-path of some previous determined path. The second simple path is combined by an edge $e(u, v)$, where vertex u is the end of the first simple path, and the SP from vertex v to destination d. Since the found deviation paths are probably not simple, their simplicity must be checked before setting as the next shortest simple path. The MPS uses a data structure called *sorted forward star form.* Denote the set of nodes V in the given diagraph $G(V, E, w)$ as $\{v_1, v_2, \ldots, v_n\}$, denote $E{=}E(v_1)\cup E(v_2)\cup\ldots\cup E(v_n)$, where $E(v_k)$ is the set of arcs whose tail node is v_k and $v_k{\in}V$. Thus, $E(v_i)\cap E(v_j){=}\emptyset$, for any v_i, $v_j{\in}E$, $i{\neq}j$. For any vertex $v_i{\in}V$, i is the index of v_i in vertex set V. Therefore, the vertices in V are sorted according to their indices. The MPS defines *sorted forward star* form as follows:

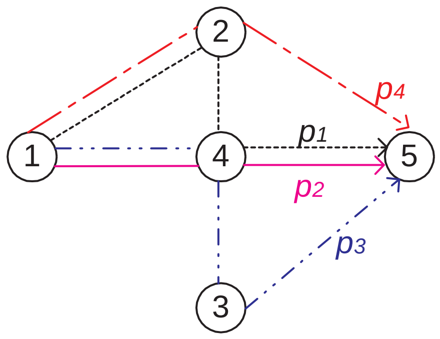
- Let e_k, $e_l{\in}E$ such that $e_k{\in}E(v_\vartheta)$ and $e_l{\in}E(v_\xi)$;
- When $\vartheta \neq \xi$, $e_k{<}e_l$, if and only if $\vartheta{<}\xi$;
- When $\vartheta{=}\xi$, $e_k{<}e_l$, if $sc(e_k){\leq}sc(e_l)$

We also can explain it in this way. For any two edges $e(v_k, v_j)$, $e(v_i, v_l){\in}E$, $(k, j){<}(i, l)$, if $k{<}i$ or ($k{=}i$, and $sc(e(v_k, v_j)){\leq}sc(e(v_i, v_l))$).

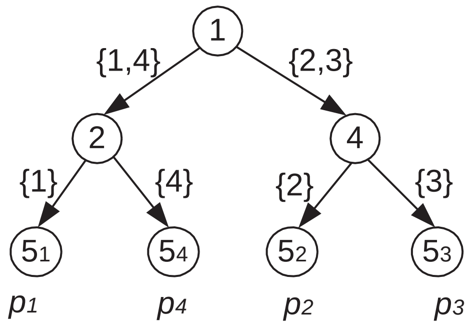
The MPS algorithm sorts the edge set of the given graph according to the sorted forward star form. Assume the resulting edge set E=$\{e_1, e_2, \dots, e_m\}$, where for any edge $e_k(1 \leq k \leq m)$, k is the position of the edge in the sequence of E. Obviously, $sc(e_k) > sc(e_{k+1})$, only if the tail node of e_k is less than the tail node of e_{k+1}. If the tail nodes of e_k and e_{k+1} are the same, then $sc(e_k) \leq sc(e_{k+1})$. For the diagraph in Figure 2.1(a), E=$\{e_{12}, e_{14}, e_{13}, e_{24}, e_{25}, e_{34}, e_{35}, e_{45}, e_{43}, e_{41}\}$ is in the sorted forward star form, where e_{ij}=$e(v_i, v_j)$ and v_i, $v_j \in E$. In the following, a concrete example is presented to explain the MPS more clearly.

Consider the given diagraph in Figure 2.1(a) and the incoming SPT T rooted at v_5 is in Figure 2.1(b). The SP p_1=$\langle v_1, v_2, v_4, v_5 \rangle$ is a path of T from vertex v_1 to root v_5. For any tree node v_i of T, denote p_{i5} as the path of T from v_i to v_5. Add p_1 to the candidate path set X. Extract the path p=$\langle v_1, v_2, v_4, v_5 \rangle$ with the minimum weight from X as p_1. The path is simple and its deviation nodes are v_1, v_2, and v_4. Analyze v_1: the edge in the sorted forward star edge set E following e_{12} is e_{14}. Then, by concatenating e_{14} with the path of T from vertex v_4 to root v_5, the deviation path of v_1: $e_{14} \diamond p_{45}$=$\langle v_1, v_4, v_5 \rangle$ is obtained and placed in X. Analyze v_2: the edge in E following e_{24} is e_{25}. Then, the deviation path is $e_{12} \diamond e_{25} \diamond p_{55}$=$\langle v_1, v_2, v_5 \rangle$, and is inserted into X. Analyze v_4: the edge in E following e_{45} is e_{43}. Then, the deviation path is $\langle v_1, v_2, v_4 \rangle \diamond e_{43} \diamond p_{35}$=$\langle v_1, v_2, v_4, v_3, v_4, v_5 \rangle$, and is placed in X. The path with the minimum weight in X is $\langle v_1, v_4, v_5 \rangle$. Since the path is simple, its is the second shortest simple path from v_1 to v_5. Although the complexity is still an open problem, the MPS performs well in practice. The experiment results demonstrate that the MPS algorithm runs much faster than Yen's algorithm.

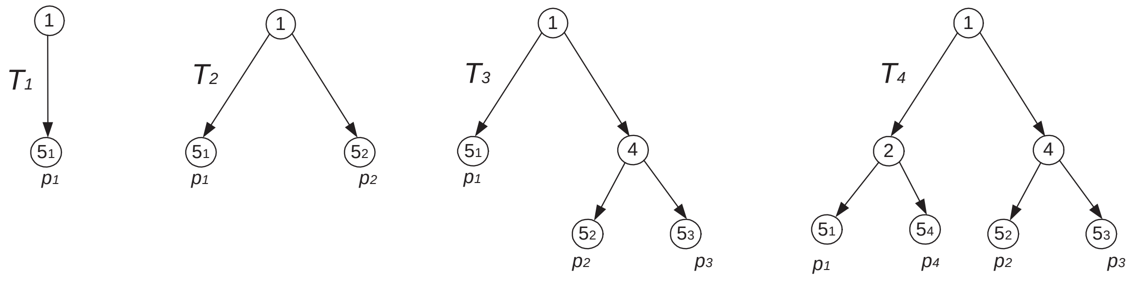
The HSB algorithm [16] proposed by John Hershberger and al. can only be adapted to directed graphs. It runs in time $O(K(m+nlogn))$ which is the best of all the algorithms for solving the K shortest simple paths problem. The HSB uses a *branching structure* to divide candidate paths set for the $(i+1)^{st}$ shortest simple path into $O(i)$ classes, and stores the shortest one in each class in a heap. The path with the minimum weight in the heap is the $(i+1)^{st}$ shortest simple path. For example, Figure 2.4(a) exhibits the first four shortest simple paths from v_1 to v_5 in G (Figure 2.1(a)), and Figure 2.4(b) reveals the branching structure of these paths. The branch from v_1 to v_2 represents edge $e(v_1, v_2, 3)$ in G, and is shared by the first and fourth paths. The branch from v_4 to v_{53} represents edge $e(v_4, v_3, 2)$ and $e(v_3, v_5, 6)$ in G, and belongs to the third path. This path branching



(a) Four shortest simple paths



(b) Path branching structure



(c) Path branching structure T_1– T_4

Figure 2.4: Example of the HSB algorithm

structure T_i is a rooted tree with source node s as its root. In T_i, we know how the first i shortest simple paths deviate from each other topologically. The property that T_{i+1} can be easily generated from T_i by some modifications is very useful to solve the problem. As a result, T_K can be constructed from T_1, T_2, ..., T_{K-1} and the K shortest simple paths are obtained from it. The first shortest simple path p_1 corresponds to T_1 with only one branch from the source to the destination representing p_1. Then the HSB computes the paths with the minimum weight deviating from the branch from v_1 to v_{51} and vertex v_1 respectively, which is $\langle v_1, v_4, v_5\rangle$ and $\langle v_1, v_2, v_5\rangle$. Then the HSB inserts them into candidate path set X. Next, the HSB extracts the path with the minimum weight from X as the second shortest simple path p_2=$\langle v_1, v_4, v_5\rangle$. T_1 is updated to T_2 as depicted in Figure 2.4(c). Since p_2 is removed from X, and deviates from v_1, another path with the minimum weight deviating from v_1 not via $e(v_1, v_4)$ and $e(v_1, v_2)$, which is $\langle v_1, v_3, v_4, v_5\rangle$, is inserted into X. The branch from v_1 to v_{52} is new. A path with the minimum weight deviating from the branch is $\langle v_1, v_4, v_3, v_5\rangle$, and is inserted into X. After the next shortest simple path is extracted, the branch structure is updated and the candidate path set X is maintained. Figure 2.4(c) reflectes the updating process of path branching structure from T_1 to T_4.

Yen's algorithm, a classical algorithm, uses the deviation concept to solve the problem efficiently. However, its running time is not competitive. According to the experimental results, the MPS outperforms Yen's algorithm and is very efficient, but the running time complexity of the MPS is unknown. The complexity of the HSB is the best of all the algorithms.

The Problem of Enumeration

The complexity of applying K SPs algorithms to solve the WCSSP is exponential [10]. In [15], G. Y Handler and I. Zang compare their implementation of Yen's algorithm with their Lagrangean relaxation approach, and reach the conclusion that their method is substantially faster, in some cases, by an order of magnitude.

2.1.2 Weight Constrained Shortest Path Problem

Except for the previously mentioned K SPs algorithms, some other approaches such as node labelling and Lagrangean relaxation can be applied to solve the WCSPP. The node labelling approach is based on dynamic programming equations, and can be found in a number of papers ([1], [9], [17]). Particularly, the LSA algorithm proposed in [9] is widely regarded as the most effective for the WCSPP [10]. The Lagrangean relaxation is another popular method applied to the WCSPP ([15], [5], [25]). Here, the idea is to divide the constraints into different levels from *easy* to *difficult* by using a Lagrangean dual multiplier to penalize the *hard* constraints. However, since the space required by the Lagrangean method is too large, it is not practical.

In this section, the LSA is introduced to solve the WCSPP in directed graph $G(V, E, w)$. Let $R{\geq}1$ be the number of weight dimensions. For each edge $e(i, j){\in}E$, $c_{ij}{\in}\Re$ is its cost and $w_{ij}{=}(w_{ij}^1, \ldots, w_{ij}^R){\in}Z_+^R$ is the weight of the edge, where w_{ij}^r indicates the r^{th} weight of the edge ($r{\in}[1, R]$). Let $W{=}(W_1, \ldots, W_R){\in}Z_+^R$ be the specified weight constraint. The weight constrained SP p from source s to destination d is the minimum cost s-d path satisfying $w_p^r \leq W_r$. The LSA attaches a label set for each vertex i in the graph such that I_i is the index set of the labels on vertex i. The labels on any vertex differ from each other. For each label $k \in I_i$, the label corresponds to a path from s to i with cost C_i^k and weight $W_i^k = (W_i^{1k}, \ldots, W_i^{Rk})$. For any other label $k' \in I_i$, if $C_i^{k'} < C_i^k$, there is at lease one weight item such that $W_i^{rk'} > W_i^{rk}$. For any two distinct labels (C_i^x, W_i^x) and (C_i^y, W_i^y) on vertex i, representing two different paths P_i^x) and P_i^y), respectively, (C_i^x, W_i^x) *dominates* (C_i^y, W_i^y), if and only if $C_i^x \leq C_i^y$ and $W_i^x \leq W_i^y$. If a label on a vertex is not dominated by any other label on the vertex, then the label and its corresponding path is *efficient*.

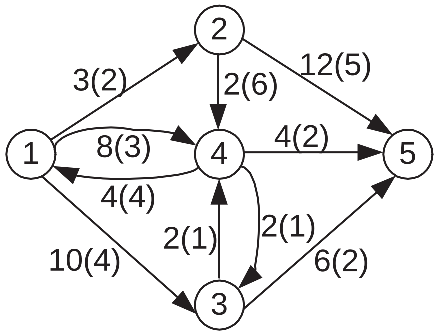


Figure 2.5: Digraph G with edges labelled by cost(weight)

The LSA is similar to Dijkstra's algorithm. The difference is that the LSA has to take care of the weight aggregation along paths in addition to the cost. The goal of the LSA is to find all the efficient labels on each vertex in the given graph. Initially, the only label is (0, 0) on s. For some total order on weights, the LSA iteratively chooses an *untreated* label with the minimum weight, and *treats* it in the following way: assuming the label is on vertex i, for each outgoing edge $e(i, j)$, a new label on j is created. If the new label (C_j^x, W_j^x) is not dominated by any other label on j, then it is added to j's label set and all the existing labels dominated by (C_j^x, W_j^x) are removed from the label set. Consequently, the first treated label is (0, 0) on s, and the set of labels is expanded during each treatment. The algorithm terminates, when all the labels of each vertex in the graph have been treated. Figure 2.5 gives a diagraph G with one dimension weight. We want to find an SP from

v_1 to v_5 with a weight less than 6. Obviously, path $\langle v_1,\ v_2,\ v_4,\ v_5\rangle$ is an SP from v_1 to v_5, but its weight is 10 which is greater than 6. The LSA computes as follows: first, treat label (0, 0) on v_1 so that label (3,2) is on v_2, label (8,3) on v_4, and label (10, 4) on v_3. The min-priority Q is ordered by (weight, cost) so that after the treatment, Q contains elements (v_2, 3, 2), (v_4, 8, 3), and (v_3, 10, 4). Extract the minimum item (v_2, 3, 2) from Q and treat it by adding label (5, 8) on v_4 and label (15, 7) on v_5. Q is updated to be (v_4, 8, 3), (v_3, 10, 4), (v_5, 15, 7), and (v_4, 5, 8). Again, extract (v_4, 8, 3) from Q and treat it. The label (9, 4) is on v_3 and label (12, 5) is on v_5. Since label (9, 4) dominates label (10, 4) on v_3, label (10, 4) is removed from the label set of v_3 and the item (v_3, 10, 4) is removed from Q. Because label (12, 5) dominates label (15, 7) on v_5, delete label (15, 7) from the label set of v_5 and item (v_5, 15, 7) from Q. Now Q contains elements (v_3, 9, 4), (v_5, 12, 5), and (v_4, 5, 8). When all the labels of each vertex are treated, the LSA terminates. An optimal path of the query is $\langle v_1,\ v_4,\ v_5\rangle$ with cost 12 and weight 6.

The complexity of the LSA is $O(|E|\prod_{r=1}^{R}(W_r{+}1))$, if the appropriate data structures are employed. When there is only one weight restriction W which is $R{=}1$, its complexity is $O(|E|\,W)$, assuming that no zero weight is allowed. In practice, the algorithm will perform well, if the value of W is not very large [10].

2.1.3 Dynamic Shortest Path Problem

The dynamic graph problem is divided into two categories based on the type of the update allowed. The *fully dynamic graph problem* allows insertion and deletion on edges, whereas the *partially dynamic graph problem* accommodates only either insertion or deletion on edges. The problem is said to be *incremental* to an edge insertion and *decremental* to an edge deletion. Another class of constrained route query in the real-world, which is to find an SP between two vertices without passing a set of edges, can be regarded as a decremental dynamic problem. After deleting the previous edge set, the SP in the modified graph is the solution to this kind of constrained query. If the SPs information in the original graph is available, these SPs can be maintained without a computation from scratch. The study of dynamic SPs maintenance began more than thirty years ago. Some of the algorithms that have been proposed to deal with this problem focus on the maintenance of the all-pairs SPs ([4], [8], [7]), whereas others ([13], [12], [26], [27]) concern the maintenance of SPTs.

In this section, we introduce the dynamic SPT algorithm (BSM) proposed in [26], which has the best performance in terms of computational complexity, as well as the minimum number of changes made to the topology of an SPT [26].

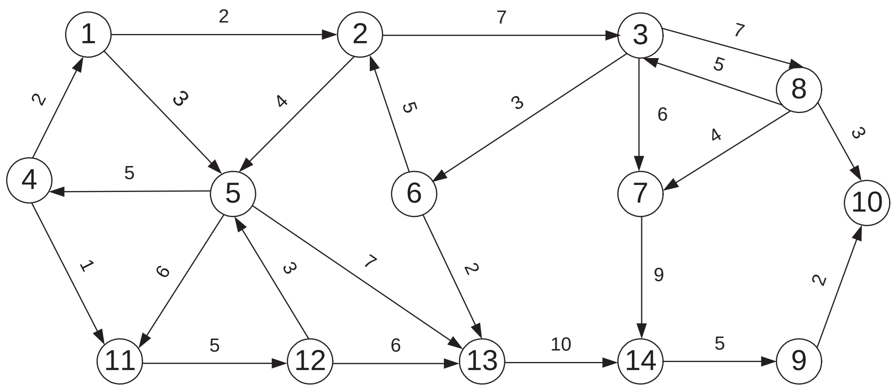
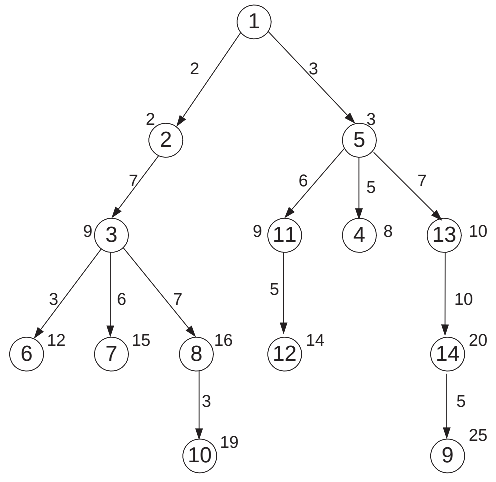


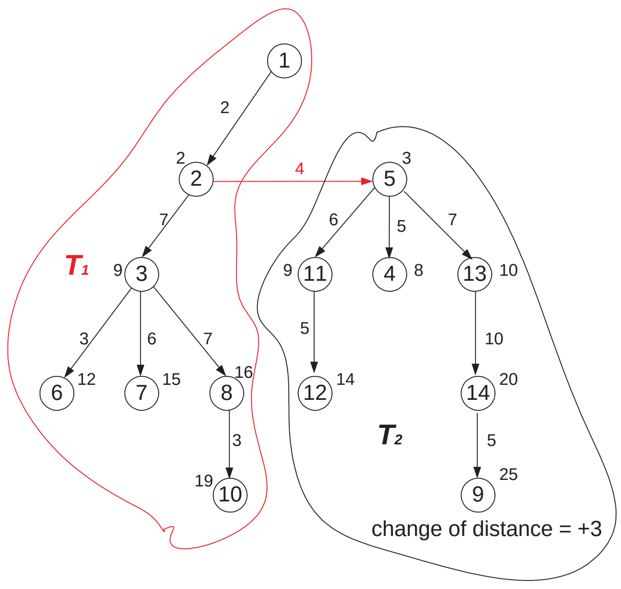
Figure 2.6: Digraph G

Dijkstra's algorithm maintains a min-priority queue Q holding vertices with its potential shortest distance from the source as a value, and iteratively, extracts a vertex with the minimum value from the queue. Also the BSM maintains such a queue, but the value of each item is the potential distance change. The BSM removes the vertex with the minimum distance change from the queue. Furthermore, Dijkstra's algorithm selects one vertex at a time, whereas the BSM selects a sub-tree from the out-dated SPT at a time. Figure 2.6 represents a digraph G, and Figure 2.7(a) gives a SPT T with v_1 as root in G. If edge $e(v_1, v_5)$ is deleted from G, e is also removed from T which is divided into two parts. The following example demonstrates how the BSM works.

Denote vertex v in Q as v, $(p, d, \triangle)$, where p is the potential predecessor of v, d is the potential distance from the source to v, and $\triangle$ is the potential distance change of v. The BSM maintains T in the following way: initially the only item in Q is $\{v_1$, (null, 0, 0)$\}$. The BSM extracts v_1 from Q, closes all the vertices in sub-tree T_1 rooted at v_1, and relax the outgoing edges of the vertices. For example, the outgoing edges of v_2 are $e(v_2, v_3)$, $e(v_2, v_5)$. When v_3 is closed, the BSM deals with only v_5. The distance of v_5 obtained from v_2 is 2+4 =6, and the change of distance is 6-3=3. Finally Q contains items $\{v_5$, $(v_2$,



(a) Outdated SPT rooted at v_1



(b) New SPT rooted at v_1

Figure 2.7: Example of the BSM algorithm

6, 3)}, {v_{14}, (v_7, 24, 4)}, and {v_{13}, (v_6, 14, 4)}. Then, the BSM extracts minimum item v_5 from Q, closes all the vertices in sub-tree T_2 rooted at v_5 to obtain a new SPT rooted at v_1. Therefore, the BSM uses only two iterations to complete the SPT maintenance. Obviously, using the BSM is much more efficient than computing a new SPT from scratch. However, for all the dynamic SP algorithms, it is noteworthy that the pre-condition is an SPT or all the SPs have been available.

2.2 Unconstrained Route Optimization

An unconstrained optimal route query is actually an SP problem. When the main memory is large enough to hold a whole graph, many algorithms such as Dijkstra's algorithm, Bellman-Ford algorithm, and Floyd-Warshall algorithm [6] can solve the problem efficiently. However, if the given graph is very large, these algorithms do not work correctly. For example, the disk file size of the graph for the east five states road system is 310M. When loading the graph into the main memory, it occupies around 1G main memory. In this section, the focus is on introducing some SP algorithms for a very large graph.

Many algorithms have been proposed to solve the SP problem within a disk-based framework. Although their approaches differ, the concept is the same. All of them depend on the divide-and-conquer technique, and consist of pre-processing and SP query evaluation phases. During the pre-processing, we first divide the graph into a set of fragments, which is small enough to be loaded into the main memory, and materialize them in some disk-based data structure, where the vertices shared by two or more sub-graphs are boundary vertices. After some pre-computation has been conducted on the partitioned graph, some resultant data is stored. During the SP query processing, the algorithms make full use of the pre-computed data in order to speed up answering the query and to reduce the I/O access. There are two materializations of the pre-computed data. One is to store the paths information between all pairs of boundary vertices in each sub-graph; the other is to materialize the distance data between the boundary vertices. The Hierarchical Encoded Path View algorithm (HEPV) [19] takes the path materialization, causing excessive storage overhead to maintain a large amount of pre-computed path information [20]. According to the experimental results in [28], materializing the shortest distances between the boundary vertices inside the same fragment provides the best savings in computation time with a given amount of storage and a small number of fragments. The algorithms, proposed in [31] (DiskSP) and [21] (DiskSPN), adopt the distance materialization.

In the pre-processing phase, [31] describes a scalable algorithm, based on breadth first search (BFS) and Hilbert R-Tree, to partition a graph into a set of fragments. Then, for each fragment, a DM holding the shortest distances between every pair of the boundary vertices in the fragment is materialized. A super graph is constructed from these DMs and boundary vertices. The nodes of the super graph are the boundary vertices, and its edges correspond to the SPs between two boundary vertices in the same fragment with their weights stored in DMDB. Since the size of the super graph is still too large, a sketch graph is proposed to capture the information of the super graph. According to the partition algorithm, a set of boundary vertices shared by two fragments is called a boundary set. Each node in a sketch graph represents a boundary set; if two boundary sets are in same fragment, then there is an edge between them in the sketch graph. Because the number of boundary sets in a fragment is small, the size of a sketch graph is also small. To find the adjacent super edges of boundary vertex v of boundary set $BS[F_i,\ F_j]$ in the super graph, the adjacent edges of $BS[F_i,\ F_j]$ in the sketch graph are found, providing all the boundary sets (including $BS[F_i,\ F_j]$ itself) in F_i and F_j. Then the boundary vertices held by the previous boundary sets are retrieved, and finally with the help of the DMs of F_i and F_j, all the information of the adjacent super edges of v are obtained. In this way, no real super graph, which almost has the same size as the original graph, needs to be constructed. After this phase, the materialized data consists of a fragment database holding all the fragments, a DMDB, a boundary set database containing boundary vertices data, and a sketch graph.

In the query processing stage, there are two steps to answer query $Q(s,\ d,\ G)$, where S and D be the fragments holding s and d, respectively. The first step is to compute an SP p from s to d in the merged graph, built by combining S and D into the super graph, with Dijkstra's algorithm. Then, for each super edge of p, find the corresponding actual path in the graph, and merge these paths into a complete path from s to d, which is the answer to query $Q(s,\ d,\ G)$.

According to [21] and [31], the relaxation of a closed boundary vertex during the computation of SP in the merged graph leads to the highest I/O costs, since the weights of the super edges are stored in the external memory. The complexity of DiskSP's I/O cost is $O(sort(N))$=$\Theta(N/B^*\ \log(M/B)(N/B))$, where N=$|V|$+$|E|$, B is the number of vertices and edges per disk block, and M is the number of vertices and edges that can be put into the internal memory [31]. This complexity is better than the best-known disk-based SP algorithm, which is $O(|V|+|E|/B)\log(|V|/B))$([3],[2]).

The DiskSPN algorithm is an improved version of the DiskSP algorithm. In the preprocessing phase, the DiskSPN pre-computes and materializes a boundary set distance matrix, containing the α-value and β-value between each pair of boundary sets in the sketch graph. Then, in the query processing phase, the DiskSPN prunes search space with the BSDM before the SP computation in the merged graph. First, the algorithm computes the β-approximation of $SD(s,\ d,\ G)$. Then, for each boundary set BS in the sketch graph, the algorithm calculates the α-approximations from s to BS and from BS to d, respectively. When the sum of two α-approximations is greater than the β-approximation, the boundary set is pruned from the sketch graph, the detailed proof is in [21]. As a result, the merged graph is also pruned. The remaining work is to compute an SP in the pruned merged graph, and fill in the actual path from each edge on the path. The SP is called *skeleton path*. The algorithm avoids a blind search and reduces the searching space in the super graph.

[21] describes how to compute $\beta A(s,\ d,\ G)$ and α-approximation between a vertex and a boundary set. $\beta A(s,\ d,\ G)$ is the minimum of the following two values:

- $min(\{\beta A(s,\ bs_s,\ S)+\alpha(bs_s,\ bs_d,\ G)+\beta A(bs_d,\ d,\ D)\ \mid\ bs_s$ is a boundary set in S and bs_d a boundary set in $D\})$;
- $min(\{\alpha A(s,\ bs_s,\ S)+\beta(bs_s,\ bs_d,\ G)+\alpha A(bs_d,\ d,\ D)\ \mid\ bs_s$ is a boundary set in S and bs_d is a boundary set in $D\})$.

The DiskSPN algorithm calculates $\alpha A(s,\ bs,\ G)$ and $\alpha A(bs,\ d,\ G)$, where bs is a boundary set in the partition of graph G, as follows:

- $\alpha A(s,\ bs,\ G)=min(\{\alpha A(s,\ bs_s,\ S)+\alpha(bs_s,\ bs,\ G)\ \mid\ bs_s$ is a boundary set in $S\})$;
- $\alpha A(bs,\ d,\ G)=min(\{\alpha(bs,\ bs_d,\ G)+\alpha A(bs_d,\ d,\ D)\ \mid\ bs_d$ is a boundary set in $D\})$.

In these equations, the α-value and β-value between the two boundary sets are retrieved from the BSDM, the α-value and β-value from s to a boundary set in S, and from a boundary set in D to d are obtained from the SPT computation in S and D. The sum of $\alpha A(s,\ bs,\ G)$ and $\alpha A(bs,\ d,\ G)$ are the lower bound of the shortest distance from s to d in G via any boundary vertex in boundary set bs. If the α-approximation via bs is greater than $\beta A(s,\ bs,\ G)$ which is an upper bound of the shortest distance from s to d in G; that is, $\alpha A(s,\ bs,\ G)+\alpha A(bs,\ d,\ G))>\beta A(s,\ d,\ G)$, then boundary set bs can be pruned. According to the experimental results in [21], the DiskSPN performs 30% better than the original algorithm DiskSP.

2.3 Proposed Disk-based Shortest Path Algorithm and Optimal Path Algorithm

Until now, little research has been conducted on the constrained route optimization problem in a very large graph. From the survey of the constrained path problem, it is evident that it is rather difficult to extend the previous K SPs algorithms or the weighted constrained path algorithms within the disk-based framework due to the large space requirement. For a dynamic SP problem, all the dynamic SP algorithms require either an existing SPT or all the SPs as the input data. In the disk-based algorithms, discussed in the last section, it is almost impossible to materialize the path information due to the scalability requirement [20]. However, for the forbidden edge route optimization problem, it is a different story. In this thesis, we attempt to solve the problem to find an SP from one vertex to another without passing a set of edges in a very large directed graph. In the following sections, we name the SP wrt the forbidden edge constrained as an *optimal path* (OP). Our work is based on the DiskSP [31] and DiskSPN [21] algorithms:

- Introduces several new improvements on the DiskSP and the DiskSPN, including the successor fragment idea to speed up the relaxation process of a closed vertex, and improved sketch graph pruning and boundary vertex pruning during the skeleton path computation. All of these ideas can be used in the new disk-based optimal path algorithm(DiskOP);

- Adopts the concept of pseudo-relaxation to balance the I/O cost and the CPU cost. Due to the forbidden edge constraint, if a fragment contains a forbidden edge, its DM is no longer valid. Two brute-force approaches address the problem: re-compute the distance matrix wrt the forbidden edge constraint, and compute an SPT wrt the previous constraint during the relaxation of each closed boundary vertex when its adjacent super edges lie in an affected fragment. The first approach minimizes the I/O cost of swapping in and out the fragments but we have to compute all pairs of the SP for boundary vertices in the fragment which increases the CPU cost. The second approach minimizes the CPU cost due to computing the SPT only when it is needed, but fragments can be swapped in and out of the main memory frequently, increasing the I/O cost. Our approach is to use the pseudo-relaxation idea to collect a set of boundary vertices of an affected fragment and relaxes their outgoing super edges inside the fragment together. This is achieved by computing one OPF, whose running time is almost the same as that of an SPT computation. In this way, the I/O cost and the CPU cost are balanced.

Chapter 3

Improvements to the Disk-Based SP Algorithms

The DiskSP algorithm [31] and its new version the DiskSPN algorithm [21] were reviewed in the last chapter. In this chapter, some improvements are proposed for the two algorithms and proved. Furthermore, some adjustments on the implementation of DiskSP and DiskSPN are described.

3.1 Relaxation in Successor Fragment

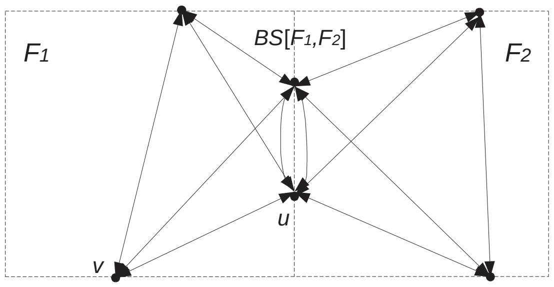
For an SP query $Q(s,\ d,\ G)$, in the query-processing phase, DiskSP applies Dijkstra's algorithm to the merged graph that consists of the source fragment, the destination fragment, and the super graph, to compute a skeleton path from s to d. DiskSP iteratively selects a boundary vertex u with the minimum shortest distance from the source vertex of all the open boundary vertices in the super graph, adds u into a closed vertex set, and relaxes the outgoing super edges of u. Let n be the number of iterations in the *while* loop of DiskSP and $BS[F_1,\ F_2]$ be the boundary set holding u, closed in the i^{th} iteration ($i \in [1,\ n]$). Obviously, all the boundary vertices in fragments F_1 and F_2 except u, are u's adjacent vertices in the super graph.

According to the definition of the super graph in Chapter 1, the weight of a super edge in the super graph is the shortest distance from the edge's head to its tail inside the fragment holding it. The triangle property for any boundary vertex x, y, and z in fragment F is $SD(x, y, F)+SD(y, z, F) \geq SD(x, z, F)$. It indicates that the shortest distance from x to z in F is less than or equal to the sum of the shortest distance from x to y and that from y to z in F. Let u be a boundary vertex in F. Assume that p is the SP from s to u by applying Dijkstra's algorithm to the merged graph. Therefore, p is a skeleton path. Let F be the predecessor fragment of u on p. With the triangle property, it can easily be proved that it is unnecessary to relax the outgoing super edges of u inside F, because the distance of a closed boundary vertex cannot contribute to the distance of any boundary vertex in its predecessor fragment. In the following sections, x.distance denotes the potential shortest distance from source s to vertex x in given graph G. Once x is closed, x.distance $= SD(s, x, G)$.

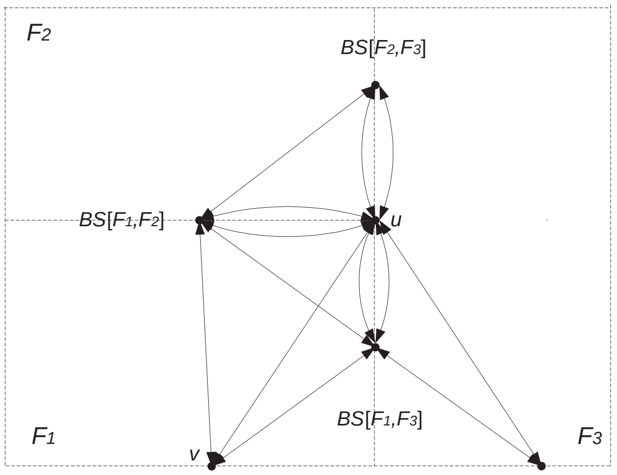
Lemma 3.1. Let vertex u be the closed boundary vertex in the i^{th} iteration of the DiskSPNN algorithm (as in Algorithm 3.4), where $i \in [1, n]$ and n is the number of iterations in the algorithm. Denote p as the skeleton path from source s to u in the super graph. Assume that F_1 is the predecessor fragment of u on p. Then, it is unnecessary to relax any super edge of u inside F_1.

Proof Assume that boundary vertex v is the predecessor of u on skeleton path p, and v is closed in the j^{th} iteration. Obviously, fragment F_1 is the successor fragment of v on p, and $1 \leq j < i$. Because the super edge $e(v, u, F_1)$ is relaxed only when v is closed, all the outgoing super edges of v inside F_1 has been relaxed in the j^{th} iteration. Thus, during the relaxation process of v in the j^{th} iteration, for any boundary vertex x in F_1, if $SD(s, v, G)+SD(v, x, F_1) < x$.distance , then x.distance is updated to $SD(s, v, G)+SD(v, x, F_1)$. As a result, at the end of the j^{th} iteration, x.distance$\leq SD(s, v, G)+SD(v, x, F_1)$. Since $j<i$, in the i^{th} iteration, we still have x.distance$\leq SD(s, v, G)+SD(v, x, F_1)$. According to the triangle property, $SD(v, x, F_1) \leq SD(v, u, F_1)+SD(u, x, F_1)$. Since $SD(s, u, G)=SD(s, v, G)+SD(v, u, F_1)$, x.distance$\leq SD(s, v, G)+SD(v, x, F_1)=SD(s, u, G)$-$SD(v, u, F_1)+SD(v, x, F_1) \leq SD(s, u, G)+SD(u, x, F_1)$. Consequently, u is not the predecessor of any boundary vertex x in its predecessor fragment F_1 on the skeleton path from s to x. It is unnecessary to relax any super edge of u inside F_1.$\square$

Assume $BS[F_1,\ F_2]$ is the only boundary set holding the above boundary vertex u. When u is closed, DiskSP and DiskSPN relax the outgoing super edges of u inside both F_1 and F_2. According to *Lemma 3.1*, let F_1 be the predecessor fragment of u on the skeleton path from s to u. Relaxing the super edges inside F_2 is enough. If u is the predecessor of boundary vertex x on skeleton path p' from s to x, where x is in F_2, F_2 is the successor fragment of u and the predecessor fragment of x on p'. F_2 is called a *potential successor fragment* of vertex u.



(a) Boundary vertex shared by two fragments



(b) Boundary vertex shared by three fragments

Figure 3.1: Relaxation on super edges

If more than two fragments, without the loss of generality, assume that fragment F_1, F_2, and F_3, hold boundary vertex u, then u has three copies u_1, u_2, and u_3 in boundary set $BS[F_1,\ F_2]$, and $BS[F_1,\ F_3]$, $BS[F_2,\ F_3]$, respectively (Figure 3.1). Let u_1, u_2, and u_3 be the first, second, and third copy to be closed. For both u_1 and u_2, assume that boundary vertex v is the predecessor, and F_1 is the predecessor fragment on the skeleton paths from s to u_1 and u_2, respectively. According to *Lemma 3.1*, relaxing the outgoing super edges of u_1 inside F_2 is enough. Since u_3 belongs to $BS[F_2,\ F_3]$, u_3.distance is updated to $SD(s,\ u_1,\ G)$, u_1 is u_3's potential predecessor, and F_2 is u_3's potential predecessor fragment on the skeleton path from s to u_3. Similarly, the relaxation process of u_2 is inside F_3 only. Finally, with closing u_3, the relaxation on u_3's super edges is inside F_3. DiskSP and DiskSPN relax the outgoing super edges for each copy of the boundary vertex in both the predecessor and successor fragments. In this example, F_2 and F_3 are the potential successor fragments of u. From the previous analysis, it is evident that even if boundary vertex u is shared by more than two fragments, only the outgoing super edges of u inside potential successor fragments are relaxed.

With the successor fragment concept, the speed of the relaxation process will be increased. Since DiskSP retrieves the adjacent super edges of a boundary vertex by a sketch graph and a DMDB, stored in the external memory, the I/O cost of reading the DMs is reduced to 50% by reading one DM instead of two DMs. The same idea can be applied to solve the disk-based OP problem. For any super edge $e(u, v, F)$ inside unaffected fragment F, the edge's weight remains as its corresponding shortest distance, $w(e)=SD(u, v, F)$. For any super edge $e(u, v, F)$ inside affected fragment F, the edge's weight is the optimal distance from u to v in F and $w(e)=OD(u, v, F, E_f)$, where E_f is the forbidden edge set given in the optimal route query. According to the definition, the optimal distance is the shortest distance in the modified graph by removing all the forbidden edges. Consequently, the successor fragment idea is still valid to the disk-based OP problem.

3.2 Query Super Graph

For an SP query $Q(s, d, G)$, the DiskSP [31] algorithm directly computes a skeleton path from s to d in the merged graph consisting of the source fragment, the destination fragment, and the super graph. With DiskSPN [21], the merged graph is pruned first, and then the skeleton path is computed in the pruned merged graph. During the pruning phase of DiskSPN, in order to calculate the β-approximation and the α-approximations, an outgoing SPT SPT_s rooted at source s in source fragment S and an incoming one SPT_d rooted at destination d in destination fragment D are computed. Thus, the shortest distance from s to any vertex of S inside S and those from any vertex of D to d inside D can be directly obtained from SPT_s and SPT_d. However, when computing a skeleton path in the pruned merged graph, DiskSP and DiskSPN must read the interior vertex or edge data of S and D and re-calculate on them. Our approach is to use the query super graph of $Q(s,\ d,\ G)$ to replace the merged graph and compute a skeleton path from s to d in the query super graph. In this way, we can ignore the interior vertices and edges in S and D.

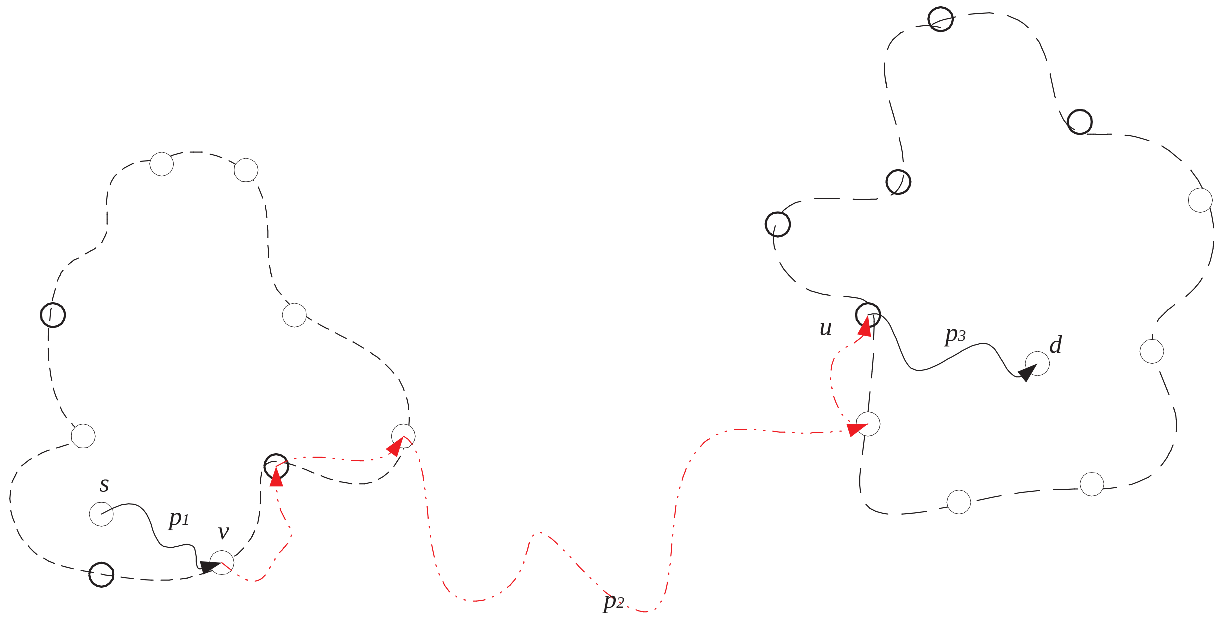


Figure 3.2: Skeleton path with s and d in the different fragments

According to [31], skeleton path p from s to d in the merged path consists of three parts (Figure 3.2 and Figure 3.3(a)):

- An SP p_1 from s to the first boundary vertex v on p, which is a boundary vertex of S;
- An SP p_2 from v to u in the super graph, where u is the last boundary vertex on p which is a boundary vertex of D;
- An SP p_3 from u to d inside D.

The exception case is portrayed in Figure 3.3(b): here, s and d are in the same fragment, and $SD(s,\ d,\ S)$ is equal to $SD(s,\ d,\ G)$, p is an SP from s to d inside S. Since p_1 and

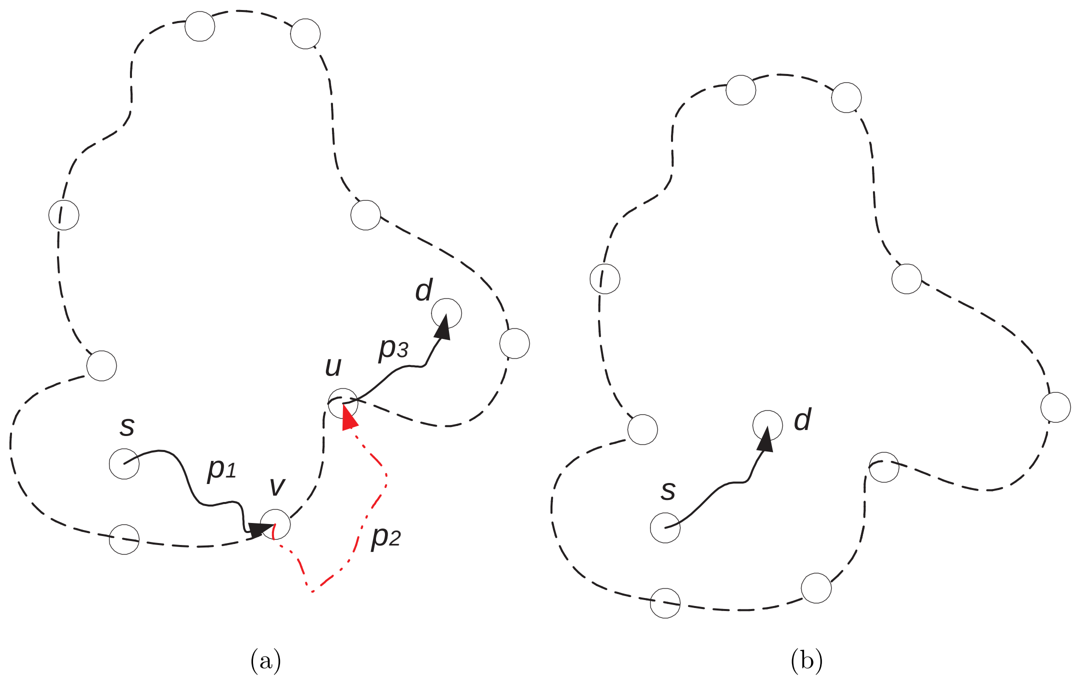


Figure 3.3: Skeleton path with s and d in the same fragment

p_3 are inside S and D respectively, they are real paths in G. However, each edge of p_2 corresponds to a super edge in the super graph. An actual path for each edge must be found. After p_1 and p_3 are merged with these paths, we obtain the answer to the SP query $Q(s,\ d,\ G)$.

Now the construction of a query super graph is briefly described. Given an SP query $Q(s,\ d,\ G)$, SPT_s and SPT_d, we can build a query super graph for the query as follows:

- Add s and d into the super graph;
- For each boundary vertex v in S, insert a super edge $e(s,\ v,\ S)$ with weight $w(e)$=$SD(s,\ v,\ S)$obtained from SPT_s, into super graph;
- For each boundary vertex u in D, insert a super edge $e(u,\ d,\ D)$ weight $w(e)$=$SD(u,\ d,\ D)$ obtained from SPT_d, into super graph;

- When s and d are in the same fragment, insert a super edge $e(s,\ d,\ S)$ with weight $w(e)$=$SD(s,\ d,\ S)$ obtained from SPT_s, into super graph.

DiskSP and DiskSPN use a sketch graph to obtain the data of a super graph. Similarly, we build a query sketch graph on a sketch graph to retrieve data of a query super graph efficiently. First, add s and d with the adjacent edges from s to each boundary set in S and from each boundary set in D to d, respectively, into the sketch graph; if S and D are the same, add an edge from s to d. The detailed definition of query super graph and query sketch graph are in Chapter 1, where the only difference is that the weight of an edge in the query super graph corresponds to the optimal distance, whereas here the weight is the shortest distance. Actually, if there is no forbidden edge constraint, the optimal distance is equal to the shortest distance. For a merged graph and a query super graph, the differences lie in the source fragment and the destination fragment. The former includes all data in S and D, whereas the latter one contains all the outgoing super edges of s inside S and the incoming super edges of d inside D. Let n_{Sb} and n_{Db} be the number of the boundary vertices of S and D, respectively. Denote $n_S(m_S)$ as the number of the vertices (edges) in S, and $n_D(m_D)$ as the number of the vertices (edges) in D. Obviously, a merged graph contains more vertices and edges than a query graph. The difference in the number of vertices is n_S-n_{Sb}+n_D-n_{Db}+2, where 2 represents the additional vertices s and d, and that in the number of edges is m_S-m_{Sb}+m_D-m_{Db}. Since it is necessary to compute SPT_s and SPT_d during the pruning stage anyway, the time to obtain the weights of the adjacent super edges of s and d can be ignored here. Consequently, computing a skeleton path in a query graph is faster than in the merged graph.

It is known that skeleton path p=p_1◇p_2◇p_3 is an SP from s to d in a merged graph. Therefore, sub-path p_1 is a real path from s to a boundary vertex v of S inside S, sub-path p_3 is a real path from a boundary vertex u of D to d inside D, and sub-path p_2 is a path consisting of the super edges and boundary vertices from v to u. By using some super edges to denote the actual SPs inside the source fragment and the destination fragment, we can ignore the details of S and D. Therefore, for an SP query, its query super graph is equal to its merged graph. Skeleton path p' from s to d in a query graph consists of a set of super edges and boundary vertices. Let $\langle e_1,\ e_2,\ \ldots,\ e_i,\ \ldots,\ e_k\rangle$ be the super edge sequence of p' and p_i be an SP corresponding to e_i, where $1 \leq i \leq k$. Then, p_1 is from s to a boundary vertex of S inside S and p_k is from a boundary vertex of D to d inside D. Thus, p_1 and p_k are obtained from SPT_s and SPT_d, respectively. For any other super edge $e_i(u, v, F)$ on p' $(1<i<k)$, its corresponding real path p_i is an SP from u to v in fragment F. Finally a resulting path p'' of the SP query $Q(s, d, G)$ can be obtained by concatenating the previous SPs in sequence, denoted as $p''=p_1\diamond p_2\diamond\ldots\diamond p_i\diamond\ldots\diamond p_k$. For the special case in Figure 3.3(b), p' has only one super edge, representing an SP from s to d inside S. Therefore, p'' is retrieved immediately from SPT_s, and is a resulting path to $Q(s, d, G)$.

3.3 Search Space Pruning

It is noteworthy that the DiskSPN algorithm [21] prunes boundary sets in a sketch graph statically. Section 3.3.1 describes an idea of pruning boundary sets by a breadth first search (BFS). Section 3.3.2 presents an approach to prune open boundary vertices during the process of a skeleton path computation.

3.3.1 Sketch Graph Pruning

The pruning technique used by the DiskSPN algorithm [21] reduces the SP search space with a pre-computed BSDM in the query-processing phase. DiskSPN computes a β-approximation $\beta A(s, d, G)$, where $\beta A(s, d, G)\geq SD(s, d, G)$. Then, for each boundary set BS in a sketch graph, two α-approximations $\alpha A(s, BS, G)$ and $\alpha A(BS, d, G)$ are computed, where $\alpha A(s, BS, G)\leq min(\{SD(s, v, G) \mid v\in BS\})$ and $\alpha A(BS, d, G)\leq min(\{SD(v, d, G) \mid v\in BS\})$. If $\alpha A(s, BS, G)+\alpha A(BS, d, G)>\beta A(s, d, G)$, then an SP from s to d in G will not pass any vertex of BS. Consequently, BS can be removed from the sketch graph. Let n_k be the number of nodes in a sketch graph. The number of α-approximation computations in the sketch graph pruning is $2n_k$ in DiskSPN.

We define three operations on a tree T with root r:

- getDistance(v): Returns the distance between r and v on T; If v is not a tree node, returns the positive infinity;

- getMinDistance(bs): Returns the minimum distance between r and a boundary vertex set bs on T which is $min(\{\text{getDistance}(v) \mid v \in bs\})$;
- getMaxDistance(bs): Returns the maximum distance between r and a boundary vertex set bs on T, which is $max(\{\text{getDistance}(v) \mid v \in bs\})$.

There are two operations on a BSDM of a partitioned graph G:

- getAlphaValue(bs_1, bs_2): Returns α-value from boundary set bs_1 to boundary set bs_2 in G which is (bs_1, bs_2, G);
- getBetaValue(bs_1, bs_2): Returns β-value from boundary set bs_1 to boundary set bs_2 in G which is (bs_1, bs_2, G).

Algorithm 3.1 AlphaApprox(r, F, T, bs, BM, out)

Input: r is a vertex, F is a fragment containing r, T is an SPT rooted at r in F, bs is a boundary set in a partitioned graph G, BM is the boundary set distance matrix, out is a Boolean value indicating if T is an outgoing tree or not.

Output: α-approximation between r and bs in G.

```
1: $min = +\infty$
2: for each boundary set $bs_1 \in F$ do
3:    if $out$ then
4:       $alpha = T$.getMinDistance($bs_1$)+$BM$.getAlphaValue($bs_1$, $bs$)
5:    else
6:       $alpha = BM$.getAlphaValue($bs$, $bs_1$)+$T$.getMinDistance($bs_1$)
7:    end if
8:    if $alpha < min$ then
9:       $min = alpha$
10:   end if
11: end for
12: return $min$
```

The AlphaApprox algorithm returns the α-approximation between vertex r and boundary set bs in the given partitioned graph. If parameter out is true, it returns $\alpha A(r,\ bs,$

G)=min($\{\alpha(r,\ bs_f,\ F)+\alpha(bs_f,\ bs,\ G)\ \mid\ bs_f$ is a boundary set in $F\}$). Otherwise, $\alpha A(bs,\ r,\ G)$=min($\{\alpha(bs,\ bs_f,\ G)+\alpha(bs_f,\ r,\ F)\ \mid\ bs_f$ is a boundary set in $F\}$).

The BetaApprox algorithm returns $\beta A(s,\ d,\ G)$ which is the minimum of the following two values:

- $min(\{\beta A(s,\ bs_s,\ S)+\alpha(bs_s,\ bs_d,\ G)+\beta A(bs_d,\ d,\ D)\ \mid\ bs_s$ is a boundary set in S and bs_d is a boundary set in $D\}$);
- $min(\{\alpha A(s,\ bs_s,\ S)+\beta(bs_s,\ bs_d,\ G)+\alpha A(bs_d,\ d,\ D)\ \mid\ bs_s$ is a boundary set in S and bs_d is a boundary set in $D\}$).

However, for those *short* queries, we observe that after pruning the boundary sets close to s, there is no path from the source to many of the boundary sets at all. Therefore, it is unnecessary to compute the two α-approximations on them. For an SP query $Q(s,\ d,\ G)$, construct a query sketch graph for it by adding s, d, and their adjacent edges into the sketch graph (details in Chapter 1). Then, traverse the query sketch graph starting from s with BFS, instead of visiting the sketch nodes one by one, and prune sketch nodes. Finally, for those unvisited sketch nodes, directly remove them from the query sketch graph. Although the worst-case complexity of the new approach is comparable to the previous one [21], when the sketch graph is large and the SP query is *short*, the new method improves the performance of pruning the query sketch graph significantly. The PruneSketchGraph algorithm is a detailed description of how to prune a query sketch graph with *BFS*. Maintain a First in First Out (FIFO) queue Q in the PruneSketchGraph algorithm. A *vector* implements a FIFO queue interface with the two operations:

- addLast(v): add an item v at the end of the FIFO queue;
- Q.removeFirst(): remove the first item from the queue and return the item.

Attach two attributes to every sketch node in the sketch graph: *visited* and *dstAlpha*. For any sketch node n, the initial value of visited attribute is *false*. Once n.visited becomes *true*, when accessing n again, we can skip it. Two α-approximations $\alpha A(s,\ BS,\ G)$ and $\alpha A(BS,\ d,\ G)$ are calculated for each visited sketch node n, where BS is the corresponding boundary set of n. The value of $\alpha A(BS,\ d,\ G)$ is stored for the later dynamic open boundary vertex pruning (details in the next section).

Algorithm 3.2 BetaApprox(s, S, SPT_s, d, D, SPT_d, BM)

Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are in respectively, SPT_s is an outgoing SPT rooted at s in S, SPT_d is an incoming SPT rooted at d in D, BM is the BSDM of G.

Output: β-approximation between r and bs in G.

```
 1: $minA = +\infty$
 2: $minB = +\infty$
 3: for each boundary set $bs_s$ in $S$ do
 4:    for each boundary set $bs_d$ in $D$ do
 5:       $betaA$=$SPT_s$.getMaxDistance($bs_s$)+$BM$.getAlphaValue($bs_s$,
          $bs_d$)+$SPT_d$.getMaxDistance($bs_d$)
 6:       $betaB$=$SPT_s$.getMinDistance($bs_s$)+$BM$.getBetaValue($bs_s$,
          $bs_d$)+$SPT_d$.getMinDistance($bs_d$)
 7:       if $betaA<minA$ then
 8:          $minA$=$betaA$
 9:       end if
10:       if $betaB<minB$ then
11:          $minB$=$betaB$
12:       end if
13:    end for
14: end for
15: if $betaA<betaB$ then
16:    return $betaA$
17: else
18:    return $betaB$
19: end if
```

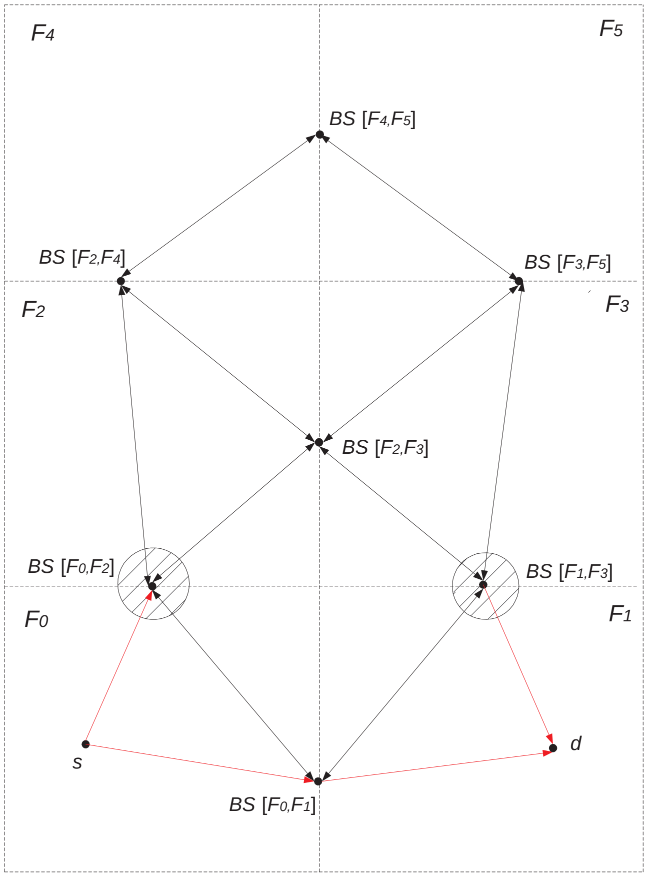
Algorithm 3.3 PruneSketchGraph(s, S, SPT_s, d, D, SPT_d, k, BM, β)

Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are in respectively, SPT_s is an outgoing SPT rooted at s in S, SPT_d is an incoming SPT rooted at d in D, BM is the BSDM, k is the sketch graph, β is $\beta A(s, d, G)$, where G is the given whole graph.

Output: none.

Ensure: sketch graph k is pruned

```
1: $Q$=new Vector() {$Q$ is a FIFO queue for sketch graph traversal (BFS)}
2: for each boundary set $bs \in S$ do
3:     $Q$.addLast($bs$)
4:     $bs$.visited=$true$
5: end for
6: while $\neg Q$.empty() do
7:     $bs$=$Q$.removeFirst()
8:     $alphas$=AlphaApprox($s$, $S$, $SPT_s$, $bs$, $BM$, $true$)
9:     $alphad$=AlphaApprox($d$, $D$, $SPT_d$, $bs$, $BM$, $false$)
10:    if ($alphas$+$alphad$)>$\beta$ then
11:        remove $bs$ from $k$
12:    else
13:        for each adjacent boundary set $bs_0$ of $bs$ in $k$ do
14:            if $\neg bs_0$.visited then
15:                $Q$.addLast($bs_0$)
16:                $bs_0$.visited=$true$
17:            end if
18:        end for
19:        $bs$.dstAlpha=$alphad$
20:    end if
21: end while
22: for each $bs$ in $k$ do
23:     if $\neg bs$.visited then
24:         remove $bs$ from $k$ {remove unvisited boundary sets in sketch graph}
25:     end if
26: end for
```



BS	αA(s, BS, G)	αA(BS, d, G)
BS[F_0, F_1]	7	4
BS[F_0, F_2]	5	12
BS[F_1, F_3]	12	3
BS[F_2, F_3]	11	9
BS[F_2, F_4]	14	15
BS[F_3, F_5]	16	12
BS[F_4, F_5]	21	18
βA(s, d, G) = 13		

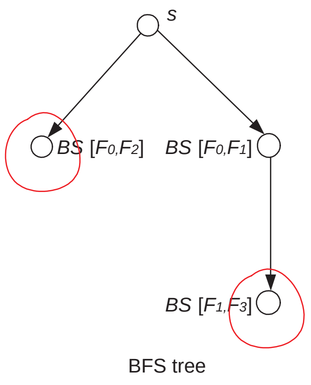


Figure 3.4: Query sketch graph pruning

Figure 3.4 provides an example of an improved query graph pruning. First, check the adjacent boundary sets of s, $BS[F_0,\ F_2]$ and $BS[F_0,\ F_1]$ in source fragment F_0, where $\alpha A(s,\ BS[F_0,\ F_2],\ G)+\alpha A(BS[F_0,\ F_2],\ d,\ G)=17>\beta A(s,\ d,\ G)=13$ and $\alpha A(s,\ BS[F_0,\ F_1],\ G)+\alpha A(BS[F_0,\ F_1],\ d,\ G)=11<\beta A(s,\ d,\ G)=13$. Hence, prune $BS[F_0,\ F_2]$. Then, check the adjacent unvisited boundary set of $BS[F_0,\ F_1]$, which is $BS[F_1,\ F_3]$, and $\alpha A(s,\ BS[F_1,\ F_3],\ G)+\alpha A(BS[F_1,\ F_3],\ d,\ G)=15>\beta A(s,\ d,\ G)=13$. Thus, $BS[F_1,\ F_3]$ is pruned. After $BS[F_1,\ F_3]$ and $BS[F_0,\ F_2]$ are removed, no path is available from s to $BS[F_2,\ F_3]$, $BS[F_2,\ F_4]$, $BS[F_3,\ F_5]$, $BS[F_4,\ F_5]$ in the sketch graph. Thus, an SP from the source to the destination will never pass through any boundary vertex of $BS[F_2, F_3]$, $BS[F_2, F_4]$, $BS[F_3, F_5]$, $BS[F_4, F_5]$. Consequently, we can prune them without computing any α-approximation. After pruning, $BS[F_0, F_1]$ is the only boundary set in the query sketch graph. With the pruning approach in DiskSPN, we need to compute 2*7 =14 α-approximations, whereas with the new method, in this example, only 2*3 = 6 α-approximations are calculated. *Lemma 3.2* proves the correctness of the PruneSketchGraph algorithm. The experimental results are discussed in Chapter 5.

Lemma 3.2. Let *bs* be a boundary set removed from k in the PruneSketchGraph algorithm, and p be an SP from s to d in graph G. Then, there does not exist a vertex v such that $v \in p$ and $v \in bs$.

Proof For any boundary set *bs*, if $\alpha A(s, bs, G) + \alpha A(bs, d, G) > \beta A(s, d, G)$, *bs* is eligible for removal from the sketch graph. Q is a FIFO queue in the algorithm. The PruneSketch-Graph algorithm traverses the query sketch graph with BFS. According to lines 13–18, there exists a path from s to any visited boundary set in the pruned query sketch graph, and the unvisited boundary sets have never been added into Q. Hence, no path is available from s to the unvisited boundary sets in the pruned query sketch graph, based on the connectivity property of a BFS tree. The PruneSketchGraph algorithm removes the boundary sets in line 11 and line 24. Therefore, any removed boundary set is either visited and satisfies the previous inequality or unvisited. In the former case, the boundary set is removed in line 11, and obviously, any SP p from s to d will not go through any of its boundary vertices. In the latter case, the boundary set *bs* is removed in line 24 and is unvisited. There is no path from s to *bs* in the pruned query sketch graph. This implies that there is no path from any visited boundary set *bs'* to *bs* such that $\alpha A(s, bs', G) + \alpha A(bs', d, G) < \beta A(s, d, G)$, according to lines 13–18. All the paths from s to *bs* in the original query sketch must pass through the nodes removed in line 11. Based on the analysis of removed visited boundary sets, $\alpha A(s, bs, G) + \alpha A(bs, d, G) > \beta A(s, d, G)$, and the lemma holds.$\square$

3.3.2 Dynamic Open Boundary Vertex Pruning

As discussed in Section 3.1, the DiskSP algorithm selects a boundary vertex or a vertex inside the source fragment or the destination fragment and relaxes the outgoing edges of the selected vertex iteratively. When DiskSP selects a boundary vertex $v \in bs$, where bs is the boundary set holding v, if v.distance+$\alpha A(bs,\ d,\ G) > \beta A(s,\ d,\ G)$, the SP from s to d will never go through v, and is eligible to be pruned. In addition, all the open vertices of bs can be pruned at this time. The cost of each boundary vertex pruning check is constant, since during the process of pruning query sketch graph, the α-approximations from those remaining boundary sets to destination d have been computed. If $\alpha A(s,\ d,\ G)$ and $\alpha A(bs,\ d,\ G)$ are tight, boundary vertex pruning will to further reduce the search space.

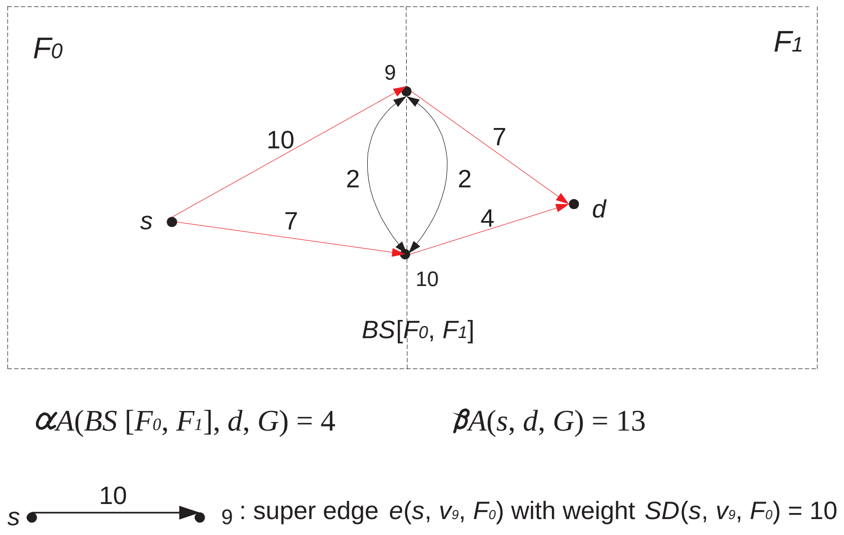

F0
F1
9
10
7
2
2
s
7
4
d
10
BS[F0, F1]
αA(BS [F0, F1], d, G) = 4
βA(s, d, G) = 13
s
10
9 : super edge e(s, v9, F0) with weight SD(s, v9, F0) = 10

Figure 3.5: Pruned query super graph

Figure 3.5 is the pruned query super graph corresponding to the pruned query sketch graph in Figure 3.4. With our approach, v_{10} is the first closed boundary vertex and its outgoing super edges are relaxed. The next closed boundary vertex is v_9. However, since $SD(s,\ v_9,\ G)+\alpha A(BS[F_0,\ F_1],\ d,\ G)=10+4>\beta A(s,\ d,\ G)=13$, prune v_9 immediately. *Lemma 3.3* provides the correctness proof of the boundary vertex pruning.

Lemma 3.3. Let vertex u be the closed boundary vertex in the i^{th} iteration of the DiskSP algorithm, where $i \in [1, n]$ and n is the number of iterations in the algorithm. Let bs be u's host boundary set. If u.distance$+\alpha A(bs, d, G) > \beta A(s, d, G)$, then u and all the open vertices in bs can be pruned.

Proof After u is closed in the i^{th} iteration, u.distance is $SD(s, u, G)$. In addition, from the definition of α-approximation, $SD(u, d, G) \geq \alpha A(bs, d, G)$. Hence, we have $SD(s, u, G) + SD(u, d, G) \geq u$.distance$+\alpha A(bs, d, G) > \beta A(s, d, G) \geq SD(s, d, G)$. Consequently, the weight of any path from s to d via u is greater than $SD(s, d, G)$; that is u is eligible for pruning. Let v be an open boundary vertex of bs in the i^{th} iteration of DiskSP. Since the DiskSP algorithm applies Diskstra's algorithm to a merged graph consisting of a super graph, a source fragment, and a destination fragment, $SD(s, v, G)$ must be greater than or equal to u.distance. Hence, $SD(s, v, G) + SD(v, d, G) \geq u$.distance$+\alpha A(bs, d, G) > \beta A(s, d, G) \geq SD(s, d, G)$. Similarly, we can prune v. □

We can also apply the idea of boundary set and boundary vertex pruning in the disk-based OP algorithms just by using γ-approximation instead of β-approximation. Because $\gamma A(s, d, G, E_f) \geq OD(s, d, G, E_f)$ is the upper bound of the optimal distance, and $\alpha A(s, bs, G) \leq min(\{SD(s, v, G) \mid v \in bs\}) \leq min(\{OD(s, v, G, E_f) \mid v \in bs\})$, $\alpha A(bs, d, G) \leq min(\{SD(v, d, G) \mid v \in bs\}) \leq min(\{OD(v, d, G, E_f) \mid v \in bs\})$. Thus, the key of the search space pruning during an OP computation is to calculate $\gamma A(s, d, G, E_f)$, which is discussed it in the next chapter.

3.4 Description of the DiskSPNN algorithm

In this section, we give the pseudo-code of the new disk-based SP algorithm (DiskSPNN) incorporating the approaches presented in the previous sections, and discuss the data structure used for the proposed algorithm, based on the data structure used in DiskSP and DiskSPN.

There are eight inputs in the DiskSPNN algorithm: source vertex s, destination vertex *d*, source fragment *S* and destination fragment *D*, fragment database *FDB*, DMDB *M*, sketch graph *k*, and BSDM *BM*. *FDB* is the partitioned graph including the whole set of fragments. The data structures *FDB*, *M* and *BM* are stored in virtual hash table, a disk-based version of the hash table, details in Chapter 4 of [31]. Each object stored in a virtual hash table has a key, can be stored or retrieved by the key. Virtual hash table maintains a buffer in the main memory. When the buffer is full and we want to access a new object, some object must be swapped onto the disk according to some strategy, and the new object is read into the main memory.

DiskSP and DiskSPN compute an SP in a merged graph, and maintain two *min*-priority queues. The vertices in *S* and *D* are in a binary heap. The boundary vertices are clustered into boundary sets, whose attributes are stored in a disk-based data structure *distance vector* [31]. Each distance vector maintains a Fibonacci heap for the boundary vertices of a boundary set with an ascendant order by the distance from *s*. A main memory U-Heap (the details are in Chapter 3 of [31]) contains one delegate vertex for each boundary set, which is the one with the minimum distance from *s* of all the open boundary vertices in the boundary set. The merit of U-Heap is that it allows the updating of the key of an entry in the heap. DiskSP selects iteratively the minimum item of all the items in the binary heap and U-Heap.

In the new DiskSPNN algorithm , since we do an SP computation in the query super graph, the binary heap is unnecessary. A U-heap and a distance vector database are enough. Moreover, after a graph is partitioned into a set of fragments, the set of boundary sets, as well as the boundary vertices in a boundary set, are fixed. Therefore, we replace a distance vector with a *distance array*. Each distance array corresponds to a boundary set, and consists of an array and a Fibonacci Heap. The Fibonacci Heap is almost the same as the one in the distance vector. The array holds the boundary vertices in a boundary set, and the vertices are sorted according to their coordinates. Each item in the array has a key as the coordinate of the boundary vertex and has a value as its auxiliary data. The auxiliary data includes four attributes: *closed*, *distance*, *predecessor* and *predecessorFrag*. The index of an item in the array is its position; for example, the index of the first item is 0 and that of the second item is 1. Closed indicates if the boundary vertex is closed or not, distance is the potential shortest distance from *s* to the current boundary vertex,

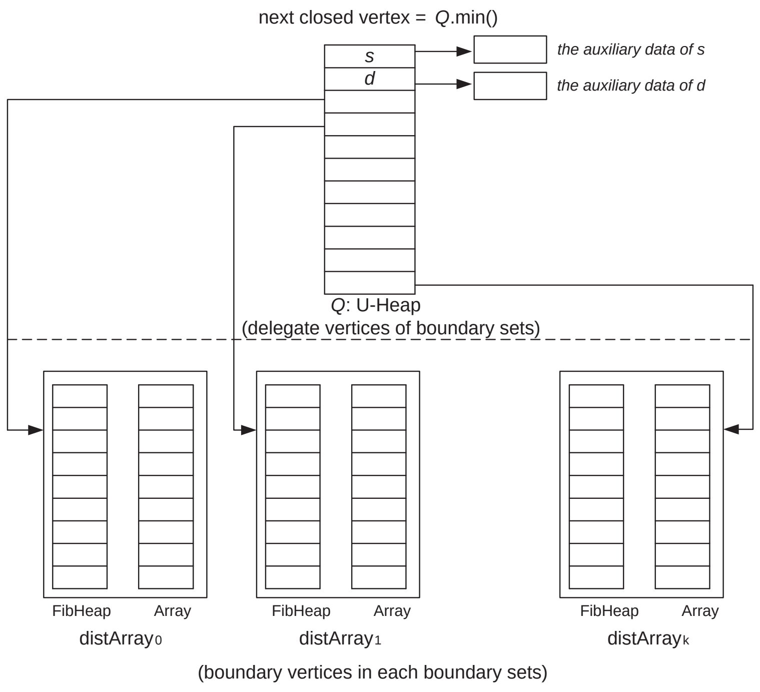

next closed vertex = Q.min()
s
d
the auxiliary data of s
the auxiliary data of d
Q: U-Heap
(delegate vertices of boundary sets)
FibHeap
Array
distArray0
distArray1
distArrayk
(boundary vertices in each boundary sets)

Figure 3.6: Data structures for DiskSPNN algorithm

predecessor is the potential predecessor of the boundary vertex, and predecessorFragment is its predecessor fragment. Therefore, we can find a boundary vertex by the ID of the boundary set holding it, and the array index of the distance array, called *boundary vertex ID*. Obviously, retrieving a boundary vertex with its boundary vertex ID is more efficient than finding the boundary vertex with its coordinate. In our new implementation, the key of any item in the U-Heap or Fibonacci Heaps is its boundary vertex ID. Since the source vertex and the destination vertex may not be boundary vertices, we need create an item for each of them in U-heap Q. In our implementation, the boundary IDs of s and d are $(BS[F_S, F_S], 0)$ and $(BS[F_D, F_D], 1)$, respectively, where F_S is the fragment ID of S, and F_D is that of D. Each of them has a pointer to the location holding its auxiliary data. Figure 3.6 exhibits the distance array database and the U-Heap. Two operations on U-heap Q are defined as follows:

- extractMin(): Returns the the item with the minimum value in the heap. It first finds the minimum item x of all the items in Q; then visits the distance array corresponding to the boundary set holding x, removes x from its Fibnaccio heap $FibQ$, and replaces x in Q with the minimum item in $FibQ$;

- update(*item*, *key*): Updates the value of an item in a distance array or in Q or according to the given key. For example, if the key of the item is a boundary vertex ID ($BS[F_i, F_j]$, index), go to the distance array representing $BS[F_i, F_j]$, and update the value of the item in the Fibnaccio heap $FibQ$ with *key*; if the minimum item is changed in $FibQ$, then replace the delegate vertex of the boundary set in Q with *item*.

In the initialization step, step A, the new algorithm computes two SPTs SPT_s and SPT_d, and computes a β-approximation for an SP from s to d in G, and prunes the boundary sets in the sketch graph. In addition, DiskSPNN initializes a distance array database to contain the distance arrays for the boundary sets during the SP computation. At this time, all the boundary vertices are open, and their distance attribute is infinity. Furthermore, DiskSPNN sets s to be closed and its distance to be zero, and relaxes the outgoing super edges of s in the query graph. In the iteration step, step B, DiskSPNN iteratively selects one vertex with the minimum value from queue Q, and relaxes the outgoing super edges of the selected vertex in its successor fragment. Since the predecessor fragment of each boundary vertex u is stored in u's auxiliary data. Let boundary set $BS[F_1,F_2]$ be u's host boundary set and F_1 be u's predecessor fragment. Thus, F_2 is the successor fragment of u. Finding the outgoing super edges of u inside fragment F_2 is easy. First obtain the adjacent node set $BSET$ of $BS[F_1,F_2]$ from the sketch graph. Then, for each boundary set bs in $BSET \cup \{BS[F_1,F_2]\}$, if bs is in fragment F_2, then DiskSPNN relaxes super edge $e(u, v, F_2)$ such that $v \in bs$. Moreover, the weights of these super edges are in the DM of F_2. When either the selected vertex is d or Q is empty, step B terminates. In the termination step, if d is closed, DiskSPNN retrieves the skeleton path from s to d from the distance array database. For each super edge $e(u, v, F)$ in the skeleton path, DiskSPNN finds a real SP corresponding to the super edge inside F with Dijkstra's algorithm, and merges these sub-paths into one complete path which is the output of the algorithm. Otherwise, d

is still open, which means no path from s to d is available in G, and the algorithm returns null.

Algorithm 3.4 DiskSPNN(s, d, S, D, FDB, M, k, BM)

Input: s and d are the source and the destination respectively, S and D are the fragments in which s and d are respectively, FDB is the fragment database of the given graph G, M is the DMDB, k is the sketch graph, BM is the BSDM.
Output: an SP from s to d in G

Step A: Initialization

```
1: Build outgoing SPT $SPT_s$ rooted at $s$ in $S$
2: Build incoming SPT $SPT_d$ rooted at $d$ in $D$
3: $\beta$=BetaApprox($s$, $S$, $SPT_s$, $d$, $D$, $SPT_d$, $BM$) {compute $\beta$-approximation from $s$ to $d$ in $G$}
4: PruneSketchGraph($s$, $S$, $SPT_s$, $d$, $D$, $SPT_d$, $k$, $BM$, $\beta$) {Prune sketch graph $k$}
5: Initialize distance array database $daDB$
6: Initialize priority queue $Q$ {$Q$ is the U-Heap containing delegates of boundary sets}
7: $s$.closed=$true$
8: $s$.distance=0
9: if $S$==$D$ then
10:    $d$.distance=$SPT_s$.getDistance($d$)
11:    $d$.predecessor=$s$
12:    $d$.predecessorFrag=$S$
13:    $Q$.update($d$, $d$.distance)
14: end if{relax the super edge $e(s, d, S)$}
15: for each boundary set $b$ in $S$ do
16:    if $k$.contains($b$) then
17:       for each boundary vertex $v \in b$ do
18:          $v$.distance=$SPT_s$.getDistance($v$)
19:          $v$.predecessor=$s$
20:          $v$.predecessorFrag=$S$
21:          $Q$.update($v$, $v$.distance)
22:       end for{relax the outgoing super edges of $s$}
23:    end if
24: end for
```

(continued next page)

```
Step B: Iteration
 1: while ¬$Q$.empty() do
 2:    $v$=$Q$.extractMin()
 3:    $v$.closed=$true$
 4:    if $v$==$d$ then
 5:       goto Step C
 6:    end if
 7:    find the boundary set $bs$=[$F_1$, $F_2$] in which $v$ locates {$bs$.dstAlpha is $\alpha A$($bs$, $d$, $FDB$)
       computed in Step A line 4}
 8:    if ($v$.distance+$bs$.dstAlpha)> $\beta$ then
 9:       prune $v$ and open vertices in $bs$
10:       remove $bs$ from $k$
11:    end if
12:    find successor fragment $F$ of $v$ {$F$ is either $F_1$ or $F_2$}
13:    MainThrust($v$, dist, $k$, $Q$, daDB, $M$, $F$) {relax the outgoing super edges of $v$ in $F$}
14: end while

Step C: Termination
 1: if $d$.closed then
 2:    construct the complete SP $p$ from the obtained skeleton path{details in [31], use
       Dijsktra's algorithm to compute a real path for each edge of the previous computed
       skeleton path and merge them into one}
 3: else
 4:    $p$=$null$
 5: end if
 6: return $p$
```

Algorithm 3.5 MainThrust(u, $dist$, k, Q, $daDB$, M, F_0)

Input: u is the closed boundary vertex, dist is the optimal distance of u, k is the sketch graph, Q is the U-Heap containing delegates of boundary sets, $daDB$ is the distance array database, M is the DM, F_0 is the successor fragment of u

Output: none.

Ensure: the outgoing super edges of u in F_0 are relaxed

```
1:  $m$=$M$.get($F_0$) {get the DM for fragment $F$ from fragment database}
2:  find all adjacent boundary sets $B$ of $bs$ from $k$ and add $bs$ into $B$
3:  for each boundary set $b \in B$ do
4:    if $b$.inFragment($F_0$) then
5:      for each boundary vertex $v \in b$ do
6:        if $v$.distance>$dist$+$m$.get($u$,$v$) then
7:          $v$.distance=$dist$+$m$.get($u$,$v$)
8:          $v$.predecessor=$u$
9:          $v$.predecessorFrag=$F_0$
10:         $Q$.update($v$, $v$.distance)
11:       end if
12:     end for
13:   end if
14: end for
```

Chapter 4

Design of the OP Query Engine

The new disk-based SP (DiskSPNN) algorithm was discussed in the last chapter. An efficient disk-based OP (DiskOP) algorithm to find an OP not via a set of edges in a very large graph is now introduced. Section 4.1 describes how to prune the search space with some pre-materialized data, Section 4.2 reviews two brute-force approaches to solve the OP problem in a very large graph, and Section 4.3 proposes our disk-based OP algorithm.

4.1 Search Space Pruning

The pruning on the sketch graph and open boundary vertices of the DiskSPNN algorithm can be easily applied to prune the search space during an OP computation in a very large graph. A pre-computed BSDM is adopted to calculate an upper bound of the shortest distance efficiently. Unlike the SP queries, it is impossible to obtain an upper bound of the optimal distance directly from BSDM due to the forbidden edge constraint. In short, the β-approximation of the shortest distance in the diskSPNN algorithm is not an upper bound of the optimal distance any longer. Fortunately, since an OP is actually an SP in the modified graph, obtained by removing those forbidden edges from the original graph, the α-approximation is still a lower bound of the optimal distance. Intuitively, to tighten α-approximation according to the forbidden edge constraint, those affected fragments must be read into the main memory, which is too time-consuming. Therefore, the

α-approximation is also used in the OP problem. As a result, what we have to solve is how to efficiently compute a γ-approximation of the optimal distance. Our approach is to materialize a pivot fragment for each fragment of the given partitioned graph and construct a query sketch graph K_Q for a given OP query $Q(s,\ d,\ G,\ E_f)$, detailed in Chapter 1. Thus, the shortest distance from s to d in the query sketch graph is a γ-approximation for $OD(s,\ d,\ G,\ E_f)$, denoted as $\gamma A(s,\ d,\ G,\ E_f)$. Section 4.1.1 gives four different approaches to materialize a pivot fragment database. The algorithm to compute the γ-approximation from s to d in G wrt E_f are discussed in Section 4.1.2. Section 4.1.3 presents the details of adapting the pruning techniques for the disk-based OP.

4.1.1 Materializations of Pivot Fragments

The DM of an affected fragment is invalid to answer an OP query $Q(s,\ d,\ G,\ E_f)$ due to the forbidden edges inside the fragment. It is impossible to obtain a tight upper bound of the optimal distance $OD(s,\ d,\ G,\ E_f)$ without the reading the interior data of those affected fragments. However, to compute the γ-approximation by reading all of the affected fragments into the main memory is not practical due to the high I/O cost. Therefore, partial fragment materialization is a better alternative to speed up the calculation process.

Since a boundary set consists of those vertices shared by two adjacent fragments, intuitively, it is assumed that any two boundary vertices in the same boundary set are close to each other. Therefore, we choose a boundary vertex randomly as the pivot of a boundary set in fragment F and materialize some paths between two pivots inside F. Then, merge all the edges of the materialized paths into a graph, which is the pivot fragment of F, denoted as F_p. Therefore, the optimal distance between any pivot pair $(pv_i,\ pv_j)$ in F_p is an upper bound of the optimal distance between them in F, that is, $OD(pv_i, pv_j,\ F_p,\ E_f) \geq OD(pv_i,\ pv_j,\ F,\ E_f)$. $OD(pv_i,\ pv_j,\ F_p,\ E_f) = PD(v_i,\ v_j,\ F,\ E_f)$ is the weight of sketch edge $e(v_i,\ v_j,\ F_p)$ in K_Q, where pv_i and pv_j are the pivots of sketch nodes v_i and v_j, respectively. During the process of computing an SP p from s to d in a query sketch graph K_Q, when relaxing the outgoing sketch edges of a sketch node inside an unaffected fragment, the weights of the edges can be read directly from the sketch graph. However, if its outgoing sketch edges are inside an affected fragment F and their weights have not been updated yet, we have to compute the optimal distances between each pivot pair in F_p as the new weights of the corresponding sketch edges. Since the size of F_p is small and the number of the pivots in F is that of the boundary sets in F, the cost of computing the weights of the sketch edges inside F is affordable. In the worst case, if there is no path from s to d in K_Q, then $\gamma A(s, d, G, E_f)$ is infinity and we cannot prune the search space at all.

There are many approaches to build a pivot fragment. This section presents four different methods to materialize a partial fragment: *single pivot path* (SPP), *double pivot path* (DP), *single branch pivot path* (SBP), and *double branch pivot path* (DBP). In Chapter 5, we compare the performance of these methods, which shows that the DBP is the best of all methods. In the rest of this chapter, the pivot fragment database that is used is materialized by DBP.

Algorithm 4.1 SPP(F, ps)

Input: F is a fragment holding u and v, ps is a set of pivots in F.

Output: pivot fragment F_p of F with SPP

```
1:  initialize a graph $F_p$
2:  for each pivot $u$ in $ps$ do
3:     for each pivot $v$ in $ps$ ($v \neq u$) do
4:        compute a SP $p$ from $u$ to $v$ in $F$
5:        for each edge $e$ of $p$ do
6:           if $\neg F_p$.contains($e$) then
7:              $F_p$.add($e$)
8:           end if
9:        end for
10:    end for
11: end for
12: return $F_p$
```

Pivot fragment F_p of fragment F consists of all the edges of the pivot paths. One or more pivot paths are computed for each sorted pivot pair in F, and each edge of the pivot paths is added into F_p. A vertex in F_p is either a pivot or an *internal* vertex. If the number of the computed pivot paths of a pivot pair is one, the materialization approach is SPP; if

Algorithm 4.2 DP(F, ps)

Input: F is a fragment holding u and v, ps is a set of pivots in F.

Output: pivot fragment F_p of F with DP

```
1: initialize a graph $F_p$
2: for each pivot $u$ in $ps$ do
3:    for each pivot $v$ in $ps$ ($v \neq u$) do
4:       compute a SP $p_1$ from $u$ to $v$ in $F$
5:       for each edge $e$ of $p_1$ do
6:          if $\neg F_p$.contains($e$) then
7:             $F_p$.add($e$)
8:          end if
9:       end for
10:      $p_2$=SecondPivotPath($u$, $v$, $F$, $p_1$) {compute a second pivot path from $u$ to $v$ in $F$}
11:      for each edge $e$ of $p_2$ do
12:         if $\neg F_p$.contains($e$) then
13:            $F_p$.add($e$)
14:         end if
15:      end for
16:   end for
17: end for
18: return $F_p$
```

it is two, the approach is DP. The materialized pivot paths of SPP are the SPs inside F. In DP, for a pivot pair (pv_i, pv_j), first path p_1 is an SP from pv_i to pv_j in F. The second path p_2 is computed in the following way: let p_1=$\langle e_1, e_2, \ldots, e_l \rangle$, check edges of p_1 in order: for each edge e_x ($1 \leq x \leq k$), if there exists a path from pv_i to pv_j in F not via e_x, delete e_x from F. Then, compute an SP from pv_i to pv_j in the modified F, which is the second pivot path. Finally, recover the modified fragment to be its original status by inserting those removed edges back. Algorithm 4.3 describes the way to compute the second pivot path in detail. Intuitively, the second pivot path tries to avoid the edges overlapping with the first pivot path. Consequently, for a set of forbidden edges, the probability of finding a tight upper

Algorithm 4.3 SecondPivotPath(u, v, F, p_1)

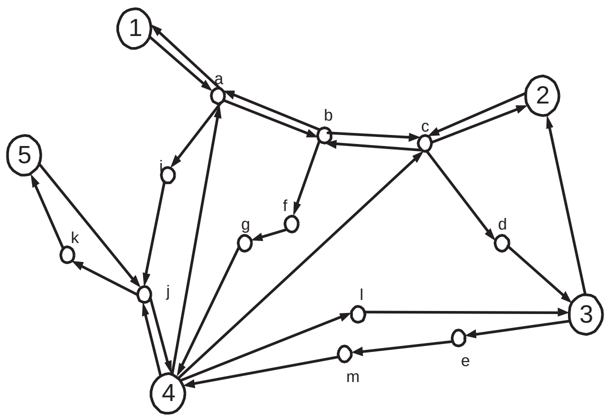
Input: u and v are pivots, F is a fragment holding u and v, p_1=$\langle e_1, e_2, \ldots, e_k\rangle$ is a SP from u to v in F.

Output: a second pivot path from u to v in F

```
1: for $i$=1 to $k$ do
2:    $F$.remove($e_i$)
3:    if there is no path from $u$ to $v$ in $F$ then
4:       $F$.add($e_i$)
5:    end if
6: end for{check edge sequence $\langle e_1, e_2, \ldots, e_k\rangle$ of $p_1$}
7: compute a SP $p_2$ from $u$ to $v$ in $F$
8: for $i$=1 to $k$ do
9:    if $\neg F$.contains($e_i$) then
10:       $F$.add($e_i$)
11:    end if
12: end for{recover the removed edges of $p_1$ in $F$, such that $F$ remains unchanged}
13: return $p_2$
```

bound for a pivot path in a fragment is increased. Algorithms 4.1 and 4.2 describe SPP and DP, respectively.

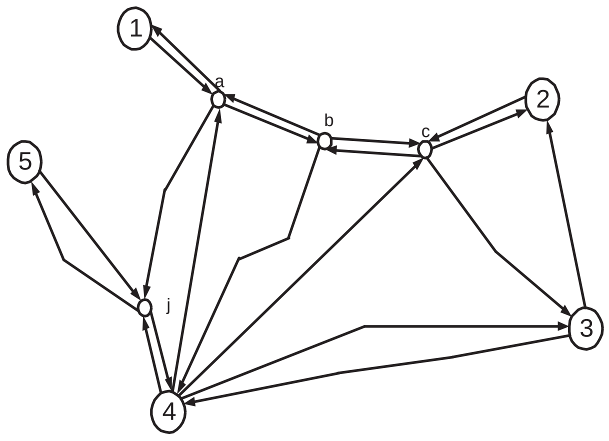
Figure 4.1 gives an example of the SPP, where v_1,v_2, v_3, v_4, v_5 are the pivots; the others are internal vertices. An internal vertex with the incoming degree 1 and outgoing degree 1 is said to be a 1-1 internal vertex. When the adjacent edges of all the 1-1 internal vertices are merged into one *branch*, the number of vertices in the pivot fragment decreases, and there is no data loss, because the branch consists of the merged edges. Branch *br* has three properties: a *tail* vertex, a *head* vertex, and an *edge ID* set, denoted as *tail*(br), *head*(br), and *edgeIDs*(br), respectively. Edge $e_i(u,v)$ and branch $b_r(v, y)$ can be merged into a new branch with head u, tail y, and edge ID set *edgeIDs*(br)$\cup\{e_i\}$. The branch idea is applied to SPP and DP to obtain two new approaches: *single branch pivot path* (SBP) and *double branch pivot path* (DBP). Figure 4.2 illustrates the pivot fragment with SBP for the example in Figure 4.1. There are 17 vertices with an SPP materialization and 9



Pivot path vertex sequence:

$p_{12} = <v_1, a, b, c, v_2>$	$p_{21} = <v_2, c, b, a, v_1>$
$p_{13} = <v_1, a, b, c, d, v_3>$	$p_{31} = <v_3, v_2, c, b, a, v_1>$
$p_{14} = <v_1, a, i, j, v_4>$	$p_{41} = <v_4, a, v_1>$
$p_{15} = <v_1, a, i, j, k, v_5>$	$p_{51} = <v_5, j, v_4, a, v_1>$
$p_{23} = <v_2, c, d, v_3>$	$p_{32} = <v_3, v_2>$
$p_{24} = < v_2, c, b, f, g, v_4>$	$p_{42} = <v_4, c, v_2>$
$p_{25} = <v_2, c, b, a, i, j, k, v_5>$	$p_{52} = < v_5, j, v_4, c, v_2>$
$p_{34} = <v_3, e, m, v_4>$	$p_{43} = <v_4, l, v_3>$
$p_{35} = <v_3, e, m, v_4, j, k, v_5>$	$p_{53} = < v_5, j, v_4, l, v_3>$
$p_{45} = <v_4, j, k, v_5>$	$p_{54} = < v_5, j, v_4>$

Figure 4.1: Pivot fragment with SPP



Internals with incoming degree 1 and outgoing degree 1: *i*, *k*, *g*, *f*, *m*, *e*, *d*.

Merge edges (a, i) (i, j) into branch(a, j);
Merge edges (j, k) (k, v_5) into branch(j, v_5);
Merge edges (b, f) (f, g) (g, v_4) into branch(b, v_4);
Merge edges $(v_3, e)(e, m)(m, v_4)$ into branch(v_3, v_4);
Merge edges $(c, d)(d, v_3)$ into branch(c, v_3)

Figure 4.2: Pivot fragment with SBP

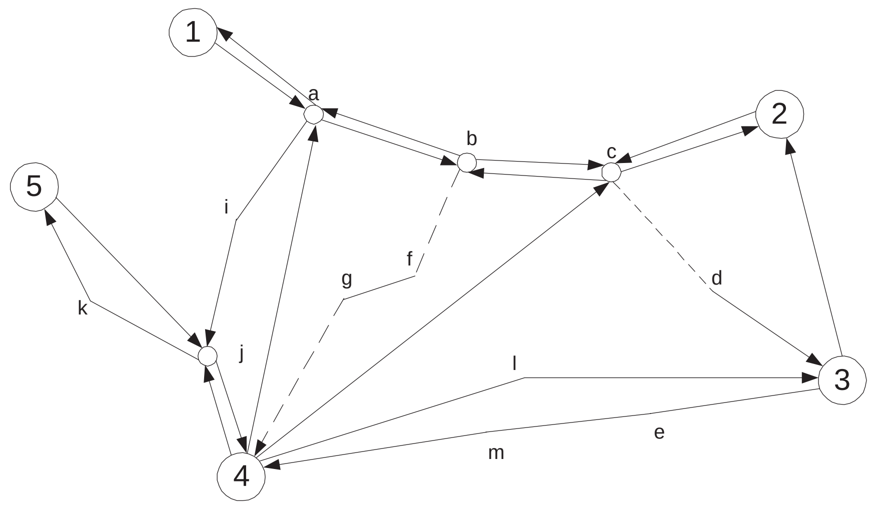


Figure 4.3: Forbidden objects of a pivot fragment with SBP

vertices with an SBP materialization without any precision or efficiency loss. Algorithm 4.4 describes the details of applying the branch idea into a pivot fragment.

Let F be an affected fragment, F_p be its pivot fragment, and E_f be a given forbidden edge set of partitioned graph G. If any edge or branch in F_p contains a forbidden edge, it is a forbidden object. Let an object o be an edge, a vertex, or a branch in graph g. There are three operations on o in g:

- contains(o): if g contains o, return *true*; otherwise, *false*;
- add(o): insert o into g;
- remove(o) : if o is an edge or a branch, then remove o from g; if it is a vertex, then remove it and its adjacent edges from g.

If it is assumed that the forbidden object set of F_p is E_{pf}, the optimal distance from pivot pv_i to pivot pv_j in F_p is the shortest distance from pv_i to pv_j in F_p not via any branch or edge in set E_{pf}, that is $PD(pv_i, pv_j, F, E_f){=}OD(pv_i, pv_j, F_p, E_{pf})$. Since F_p is a sub-graph of F, and E_{pf} is actually a subset of E_f, obviously $PD(pv_i, pv_j, F, E_f) \geq OD(pv_i, pv_j, F, E_f)$. Figure 4.3 provides an example of a forbidden object set, where the given forbidden edges

Algorithm 4.4 BranchPF(F_p)

Input: F_p is a pivot fragment with SPP or DP

Output: F_{bp} is a pivot fragment applying branch idea to F_p

```
1: new a graph $F_{bp}$, which is a copy of $F_p$
2: for each vertex $u$ in $ps$ do
3:   if $u$ is a 1-1 internal then
4:     $br$=new branch($e(x, u)$, $e(u, y)$) {$e(x, u)$ and $e(u, y)$ are $u$'s adjacent edges/branches}
5:     $F_{bp}$.remove($u$))
6:     $F_{bp}$.add($br$)
7:   end if
8: end for
9: return $F_{bp}$
```

are $e(c, d)$, $e(b, f)$, $e(g, v_4)$, and its corresponding forbidden objects are $br(c, v_3)$, $br(b, v_4)$.

4.1.2 γ-approximation

Given an OP query $Q(s, d, G, E_f)$, according to the definitions of γ-approximation and the query sketch graph in Chapter 1, the shortest distance from s to d in query sketch graph K_Q wrt $Q(s, d, G, E_f)$ is a γ-approximation from s to d in G. Given an outgoing OPT OPT_s rooted at s in S and an incoming OPT OPT_d rooted at d in D, we can build K_Q based on sketch graph K of G as follows:

- For each sketch edge $e(v_i, v_j, F)$ in K, if F is an affected fragment, then $w(e)$=$PD(pv_i, pv_j, F, E_f)$, where pv_i and pv_j are the pivots of sketch vertices v_i and v_j, respectively;
- Add s and d to the sketch graph K;
- For each boundary set bs of S in K, insert a sketch edge from s to bs with weight $OD(s, pv, S, E_f)$ into K, where pv is the pivot of bs, and $OD(s, pv, S, E_f)$ is obtained from OPT_s;

- For each boundary set bs of D in K, insert a sketch edge from bs to d with weight $OD(pv,\ d,\ D,\ E_f)$ into K, where pv is the pivot of bs, and $OD(pv,\ d,\ D,\ E_f)$ is obtained from OPT_d;
- If s and d are in the same fragment, insert a sketch edge from s to d with weight $OD(\text{s},\ d,\ S,\ E_f)$ into K, where $OD(\text{s},\ d,\ S,\ E_f)$ is obtained from OPT_s.

In the process, operation getDistance(v) is used to return the distance on a tree between root and a tree node v (detailed in Chapter 3). Essentially, the γApprox algorithm computes an SP from s to d in K_Q with Dijkstra's algorithm. Since the weights of the sketch edges inside the affected fragment are unknown, we need to compute them in real-time. When the sketch edges of a sketch node v are relaxed in affected fragment F, it is necessary to check if the weights of the sketch edges in F are computed. If not, the γApprox reads pivot fragment F_p of F from the pivot fragment database, and computes the optimal distances between each pivot pairs wrt the forbidden edges constraint in F_p. Then, the weights of the sketch edges are updated with their corresponding optimal distances. To relax the outgoing sketch edges of a sketch node in an unaffected fragment, the γApprox obtains their weights directly from sketch graph K according to the definition of the query sketch graph.

The γApprox algorithm maintains a min-priority queue Q with two operations, extractMin() and enqueue(*bs,dist*). Q.extractMin() returns the item with the minimum value in Q, and Q.enqueue(*bs,dist*) inserts item bs with value *dist* into Q. Each sketch edge $e(v_i,\ v_j,\ F)$ in K has an attribute *weight* indicating $SD(v_i,\ v_j,\ F)$. If F is affected, the edge weight is updated to be $PD(pv_i,\ pv_j,\ F,\ E_f)$ during the computation. Each sketch node bs in K has attributes: *pivot*, *distance*, and *predFrag*, indicating the pivot of its corresponding boundary set, the potential γ-approximation from s to bs in G, and the potential predecessor fragment of bs on the SP from s to bs in K_Q, respectively. Algorithm 4.5 describes the process of $\gamma A(s,\ d,\ G,\ E_f)$ calculation. *Lemma 4.1* indicates the correctness proof of the γApprox algorithm.

Lemma 4.1. $\gamma A(s,\ d,\ G,\ E_f)$ is the output of the γApprox for $Q(s,\ d,\ G,\ E_f)$.

Proof Let K be a sketch graph of partitioned graph G. Given a sketch edge $e(bs_i,\ bs_j,\ F)$ in K, according to the definition of a sketch graph, $w(e)$=$SD(v_i,\ v_j,\ F)$, where v_i and v_j

Algorithm 4.5 γApprox(s, S,OPT_s, d, D, OPT_d, k, E_f, PDB)

Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are in respectively, OPT_s is an outgoing OPT rooted at s in S, OPT_d is an incoming OPT rooted at d in D, k is the sketch graph, E_f is the forbidden edge database, PDB is pivot graph database

Output: $\gamma A(s, d, \mathrm{G}, E_f)$

Step A: Initialization

```
1: for each boundary set $bs \in k$ do
2:     $bs$.distance=$+\infty$
3: end for
4: initialize a priority queue $Q$
5: for each boundary set $bs$ in $S$ do
6:     $pv$=$bs$.pivot
7:     $bs$.distance=$OPT_s$.getDistance($pv$)
8:     $bs$.preFrag=$S$
9:     $Q$.enqueue($bs$, $OPT_s$.getDistance($pv$)) {insert boundary set $bs$ with the optimal dis-
       tance from $s$ to $pv$ in $S$ into $Q$}
10: end for{get the pivots in $S$}
11: if $S$.contains($d$) then
12:     $\gamma$=$OPT_s$.getDistance($d$)
13:     $Q$.enqueue($d$,$\gamma$)
14: else
15:     $\gamma$=$+\infty$
16: end if
```

(continued next page)

```
Step B: Iteration
 1: while $\neg Q$.empty() do
 2:    $bs[F_1,F_2]=Q$.extractMin()
 3:    if $bs$ is $d$ then
 4:       goto Step C line 1
 5:    end if
 6:    if $bs$.inFragment($D$) $\wedge$ $\gamma>$($bs$.distance+$OPT_d$.getDistance($bs$.pivot)) then
 7:       $Q$.enqueue($d$, $bs$.distance+$OPT_d$.getDistance($bs$.pivot)){relax to $d$}
 8:       $\gamma=bs$.distance+$OPT_d$.getDistance($bs$.pivot)
 9:    end if
10:    if $F_1==bs$.preFrag then
11:       $F=F_2$
12:    else
13:       $F=F_1$
14:    end if{find the successor fragment of $bs$}
15:    if $E_f$.contains($F$) $\wedge$ weights of sketch edges in $F$ not updated then
16:       for each sketch edge $e(u, v)$ in $F$ do
17:          $F_p=PDB$.get($F$) {get pivot fragment of $F$}
18:          $e$.weight=$OD(u, v, F_p, E_f)${compute the optimal distance from $u$ to $v$ in $F_p$}
19:       end for
20:    end if
21:    for each sketch edge $e(bs, bs_0, F)$ in $k$ do
22:       if $bs_0$.distance>($bs$.distance+$e$.weight) then
23:          $bs_0$.distance=$bs$.distance+$e$.weight
24:          $Q$.enqueue($bs_0$, $bs_0$.distance)
25:          $bs_0$.preFrag=$F$
26:       end if
27:    end for
28: end while

Step C: Termination
 1: return $\gamma$
```

are the pivots of boundary sets bs_i and bs_j, respectively. the γApprox algorithm constructs an augmented graph K' of K by adding s, d, and their adjacent sketch edges. For each boundary set bs_s in a source fragment S, insert a sketch edge $e(s, bs_s, S)$ with weight $SD(s, v_s, S)$ in K, and v_s is the pivot of bs_s. For each boundary set bs_d in destination fragment D, insert sketch edge $e(v_d, d, D)$ with weight $SD(v_d, d, D)$, and v_d is the pivot of bs_d. Denote p' as an SP from s to d in K'. Obviously, $w(p') \geq SD(s, d, G)$.

Let K_Q be the query sketch graph of the given query. As per the definition of K_Q, the only difference between K' and K_Q is the weights of the sketch edges inside the affected fragments. By assuming F is affected, $e(bs_i, bs_j, F)$ is with weight $OD(v_i, v_j, F_p, E_f)$, where F_p is the pivot fragment of F, and v_i, v_j are the pivots of sketch nodes bs_i, and bs_j. Because F_p is a sub-graph of F, $OD(v_i, v_j, F_p, E_f) \geq OD(v_i, v_j, F, E_f)$. Denote p to be the SP from s to d in K_Q. Because p corresponds to an actual path from s to d in G not via a forbidden edge, $w(p) \geq OD(s, \mathrm{d}, G, E_f)$, and $w(p) = \gamma A(s, d, G, E_f)$.

According to Step A lines, 5–16, the outgoing sketch edges of source vertex s in K_Q are relaxed. According to Step B, lines 6–9, all the incoming sketch edges of d in K' are relaxed. According to Step B, lines 15–20, the weights of those sketch edges wrt the affected fragments are re-computed in the corresponding pivot fragments. Therefore, K' is the same as K_Q. γApprox applies Dijkstra's algorithm to compute the SP p' in K'. According to Step B, line 21, extracted boundary set bs relaxes only in its successor fragment. *Lemma 3.1* indicates that it is unnecessary to relax the predecessor fragment. Based on the analysis, the algorithm incorporates the successor fragment idea into Dijkstra's algorithm to find the SP p' from s to d in K_Q. As a result, $w(p') = \gamma A(s, d, G, E_f) \geq OD(s, d, G, E_f)$. □

4.2 Two Brute-Force Disk-Based OP Algorithms

According to the discussion in Chapter 3, given an SP query $Q(s, d, G)$, a query super graph S_Q can be constructed. Let p be an SP from s to d in S_Q. Compute an SP for each super edge of p, and concatenate them in sequence. The obtained path is the answer to $Q(s, d, G)$. The three disk-based OP algorithms proposed in this thesis are under the same framework as the DiskSPNN algorithm described in Chapter 3; that is, to find the SP in a query super graph built for a given route query. As shown in DiskSPNN, when relaxing the outgoing edges of boundary vertex v in S_Q, the weights of the edges can be retrieved directly from the pre-computed DMDB. However, for the OP problem, the DMs of these affected fragments are invalid due to the given forbidden edge constraint; the weights of the involved affected super edges need to be computed in real time. This section introduces two straightforward approaches for solving the OP problem in a very large graph: the *Optimal Path Tree Approach* (NOPT) and the *New Distance Matrix Approach* (NDM), and discusses their drawbacks.

Both NOPT and NDM are based on the DiskSPNN algorithm which computes an SP from s to d in query super graph S_Q, corresponding to SP query $Q(s,\ d,\ G)$. The weight of each super edge $e(u,\ v,\ F)$ in S_Q is $SD(u,\ v,\ F)$. Given an optimal query Q'$(s,\ d,\ G,\ E_f)$, let S_Q' be a query super graph wrt Q'. The weight of any super edge $e(u,\ v,\ F)$ in S_Q' should be the optimal distance from u to v inside F. If F does not hold a forbidden edge, $w(e)$ remains $SD(u,\ v,\ F)$; otherwise, it is $OD(u,\ v,\ F,\ E_f)$. Thus, the SP p' from s to d in S_Q' represents an OP in G not via any edge of E_f. The NOPT algorithm and the NDM algorithm compute p' in S_Q'. When an affected super edge is relaxed, its weight must be computed; otherwise, the weight from the pre-computed DM is read. Also, NOPT and NDM incorporate the successor fragment concept and the pruning techniques introduced in Chapter 3. From *Lemmas 3.1–3.3* and *Lemma 4.1*, the correctness of the two approaches is obvious. They use the same data structure as DiskSPNN. Moreover, their initialization steps and termination steps are almost the same as DiskSPNN, except that they build OPTs instead of SPTs in a source and a destination fragment, and compute γ-approximation instead of the β-approximation. The iteration step of the two approaches must compute the weights of the affected super edges to be relaxed, and the rest is the same as that of DiskSPNN.

The NOPT algorithm applies Dijkstra's algorithm in S_Q' to find skeleton path p from a source to a destination, and then fills in each edge of p, to obtain a real path. The NOPT algorithm maintains min-priority queue Q. It iteratively extracts a boundary vertex v with the minimum optimal distance from a source vertex and relaxes its outgoing super edges inside its successor fragment F. If F does not contain a forbidden edge, then NOPT reads the weights of the super edges from the DM of F. Otherwise, the algorithm reads F into

Algorithm 4.6 NOPT(s, d, S, D, FDB, M, k, BM, E_f, PDB)

Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are respectively, FDB is the fragment database, M is the DMDB, k is the sketch graph, BM is the BSDM, E_f is the forbidden edge database, PDB is pivot graph database.

Output: an OP from s to d in FDB wrt E_f

```
Step A: Initialization
1: Build an outgoing OPT $OPT_s$ rooted at $s$ in $S$ wrt $E_f$
2: Build an incoming OPT $OPT_d$ rooted at $d$ in $D$ wrt $E_f$
3: $\gamma$=$\gamma$Approx($s$, $S$, $OPT_s$, $d$, $D$, $OPT_d$, $k$, $E_f$, $PDB$)
4: PruneSketchGraph($s$, $S$, $OPT_s$, $d$, $D$, $OPT_d$, $k$, $BM$, $\gamma$){Prune sketch graph $k$}
5: Initialize distance array database $daDB$
6: Initialize priority queue $Q${$Q$ is the U-Heap containing delegates of boundary sets}
7: $s$.closed=$true$
8: $s$.distance=0
9: **if** $S$==$D$ **then**
10:    $d$.distance=$OPT_s$.getDistance($d$)
11:    $d$.predecessor=$s$
12:    $d$.predecessorFrag=$S$
13:    $Q$.update($d$, $d$.distance)
14: **end if**{relax the super edge $e(s, d, S)$, when $s$ and $d$ are in the same fragment}
15: **for** each boundary set $b$ in $S$ **do**
16:    **if** $k$.contains($b$) **then**
17:       **for** each boundary vertex $v \in b$ **do**
18:          $v$.distance=$OPT_s$.getDistance($v$)
19:          $v$.predecessor=$s$
20:          $v$.predecessorFrag=$S$
21:          $Q$.update($v$, $v$.distance)
22:       **end for**
23:    **end if**
24: **end for**{relax the outgoing super edges of $s$ in $S$}
```

(continued next page)

Step B: Iteration

```
1: while ¬$Q$.empty() do
2:    $v$=$Q$.extractMin()
3:    $v$.closed=$true$
4:    if $v$==$d$ then
5:       goto Step C
6:    end if
7:    find the boundary set $bs$=[$F_1$, $F_2$] in which $v$ locates
8:    if ($v$.distance+$bs$.dstAlpha)>$\gamma$ then
9:       prune $v$ and open vertices in $bs$
10:      remove $bs$ from $k$
11:   end if{boundary vertex pruning, $bs$.dstAlpha is $\alpha A(bs, d, FDB)$ computed in Step
      A line 4}
12:   find successor fragment $F$ of $v${$F$ is either $F_1$ or $F_2$}
13:   if $E_f$.contains($F$) then
14:      Build an outgoing OPT $OPT_v$ rooted at $v$ in $F$ wrt $E_f$
15:      MainThrustOPT($v$, $dist$, $k$, $Q$, $daDB$, $OPT_v$, $F$)
16:   else
17:      MainThrust($v$, $dist$, $k$, $Q$, $daDB$, $M$, $F$){relax the outgoing super edges of $v$ in $F$}
18:   end if
19: end while
```

Step C: Termination

```
1: if $d$.closed then
2:    construct the complete OP $p$ from the obtained skeleton path
3: else
4:    $p$=$null$
5: end if
6: return $p$
```

Algorithm 4.7 MainThrustOPT(u, $dist$, k, Q, $daDB$, OPT_u,F_0)

Input: u is the closed boundary vertex, $dist$ is the optimal distance of u, k is the sketch graph, Q is the U-Heap containing delegates of boundary sets, $daDB$ is the distance array of boundary sets, M is the DM, OPT_u is an OPT rooted at u in F_0, F_0 is the successor fragment of u.

Output: none.

Ensure: u's outgoing super edges in F_0 are relaxed

```
1: find all the adjacent boundary sets $B$ of $bs$ from $k$ and add $bs$ into $B$
2: for each boundary set $b \in B$ do
3:   if $b$.inFragment($F_0$) then
4:     for each boundary vertex $v \in b$ do
5:       if $v$.distance>($dist$+$OPT_u$.getDistance($v$)) then
6:         $v$.distance=$dist$+$OPT_u$.getDistance($v$)
7:         $v$.predecessor=$u$
8:         $v$.predecessorFrag=$F_0$
9:         $bQ$.decreaseKey($v$, $v$.distance) {$bQ$ is the Fibonacci queue of $b$ in $daDB$}
10:        $Q$.update($v$, $v$.distance)
11:      end if
12:    end for{relax the outgoing super edges of $u$ in $F_0$}
13:  end if
14: end for
```

the main memory first, computes an outgoing OPT OPT_v rooted at v inside F, and obtains the weights of the super edges from OPT_v. Algorithm 4.7 describes the relaxation of the super edges inside an affected fragment, and algorithm 4.6 gives the description of NOPT. However, when the graph is very large and the main memory is not large enough to hold all the affected fragments, the affected fragments must be frequently swapped. As a result, the I/O cost is high. Since NOPT computes only an OPT when doing the relaxation on a closed super node with an affected successor fragment, it minimizes the CPU time for computing the OPTs.

NDM is similar to NOPT. The only difference is that NDM updates the DM according

Algorithm 4.8 NDM(s, d, S, D, FDB, M, k, BM, E_f, PDB)

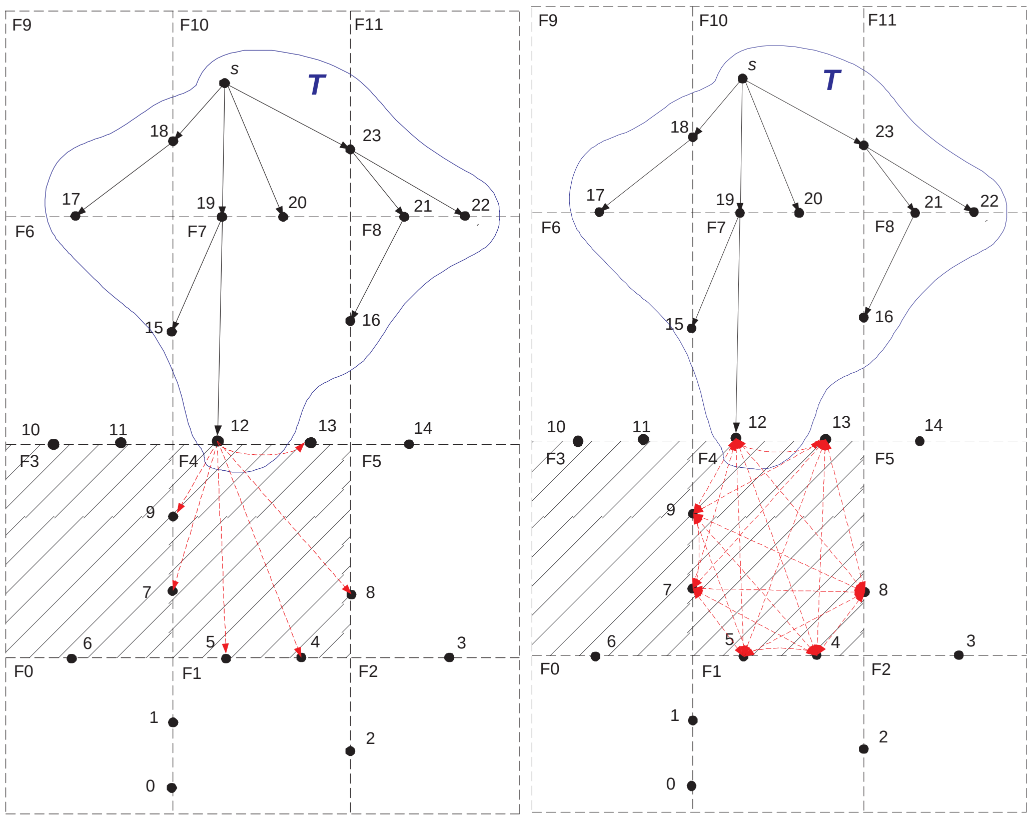
Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are respectively, FDB is the fragment database, M is the DMDB, k is the sketch graph, BM is the BSDM, E_f is the forbidden edge database, PDB is pivot graph database
Output: an OP from s to d in FDB

```
Step A: Initialization
 1: The NOPT algorithm step A
 2: initialize a new DMDB $NM$

Step B: Iteration
 1: while $\neg Q$.empty() do
 2:    $v$=$Q$.extractMin()
 3:    $v$.closed=$true$
 4:    if $v$==$d$ then
 5:       goto Step C
 6:    end if
 7:    find the boundary set $bs$=[$F_1$, $F_2$] in which $v$ locates
 8:    if ($v$.distance+$bs$.dstAlpha)>$\gamma$ then
 9:       prune $v$ and open vertices in $bs$
10:       remove $bs$ from $k$
11:    end if{prune boundary vertex, $bs$.dstAlpha=$\alpha A$($bs$, $d$, $FDB$) computed in Step A}
12:    find successor fragment $F$ of $v${$F$ is either $F_1$ or $F_2$}
13:    if $E_f$.contains($F$) then
14:       if $NDM$.contains($F$) then
15:          MainThrust($v$, $dist$, $k$, $Q$, $daDB$, $NM$, $F$){the new DM of $F$ was computed}
16:       else
17:          compute the new DM of $F$ wrt $E_f$ and put it into $NM$
18:       end if
19:    else
20:       MainThrust($v$, $dist$, $k$, $Q$, $daDB$, $M$, $F$){relax the outgoing super edges of $v$ in $F$}
21:    end if
22: end while

Step C: Termination
 1: The NOPT algorithm Step C
```

to the given forbidden edge constraint for each involved affected fragment. For extracted boundary vertex v with affected successor fragment F, NDM checks whether the DM of F has been updated or not. If the matrix is not updated, it reads F into the main memory, and computes the optimal distances between each boundary vertex pair in F, and updates the DM for F, storing it in a new DMDB *NM*. Otherwise, NDM retrieves the computed one from the *NM*. This approach reads each affected fragment into the main memory only once during the optimal skeleton path computation, minimizing the I/O cost. Let k be the number of boundary vertices in an affected fragment F. The NDM needs to compute k OPTs inside F to obtain the updated DM. Consequently, the CPU cost of NDM is high.



(a) Example of NOPT (b) Example of NDM

Figure 4.4: Two brute-force disk-based OP approaches

Figures 4.4 depicts examples of NOPT and NDM. Let F_3 and F_4 be the affected fragments in a given portioned graph. NOPT and NDM iteratively extract a super node from Q and do relaxation on its outgoing super edges inside its successor fragment. Since F_6, F_7, F_8, F_9, F_{10} and F_{11} do not contain a forbidden edge, their DMs are valid, and the weights of the super edges are directly retrieved from them. However, when v_{12} is extracted with affected fragment F_4, the weights of its outgoing super edges need to be computed. NOPT builds an outgoing OPT rooted at v_{12} in F_4 wrt the forbidden edge constraint, and obtains the weights of the edges from the tree; NDM computes an OPT for each boundary vertex in F_4 to obtain a new DM. Thus, at the time of relaxing the outgoing super edges of v_{13} in its successor fragment F_4, NOPT must build an OPT with root v_{13} in F_4. NDM attains the weights of the super edges from the updated DM of F_4. In summary, NOPT computes an OPT for a boundary vertex inside affected fragment F only if it is required; NDM computes an OPT for each boundary vertex of F. NOPT has to swap frequently the affected fragments; and NDM reads them only once. From the previous analysis, a good solution to the OP problem should balance the I/O cost and CPU time of computing the OPTs at the same time.

4.3 Proposed DiskOP algorithm

The DiskOP is also a variant of Dijkstra's algorithm. Given an OP query $Q(s, d, G, E_f)$ and query super graph S_Q, DiskOP, NOPT, and NDM compute an SP from s to d in S_Q. When closing boundary vertex v with affected successor fragment F, NOPT and NDM relax v immediately by computing an OPT with v as the root in F or calculates a new DM for F wrt E_f. However, DiskOP attempts to find a closed but un-relaxed boundary vertex set CU in F, which is the successor fragment of all the vertices in CU. In addition, CU should contain as many boundary vertices as possible that satisfy the previous restriction. Then, DiskOP relaxes the outgoing super edges of the boundary vertices in CU inside F simultaneously by computing an OPF wrt the forbidden edge set, decreasing the computation and I/O time. The reason for finding CU is to continue Dijkstra's algorithm on the graph as if no forbidden edge constraints exist, until all the unfinished vertices in F are closed. If a vertex is both closed and relaxed, it is a finished vertex. Otherwise, it is unfinished. As per the analysis in Section 3.1, let v belong to boundary set $bs[F', F]$, where bs is shared by the predecessor and successor fragments. Assume the predecessor fragment of v is F'. Then, F is its successor fragment. Obviously, the successor fragment of a super node held by a boundary set is uniquely determined. From *Lemma 3.1*, we only need to relax the outgoing super edges of v inside F. It may require a number of iterations to close all the boundary vertices in an affected fragment. The details of this approach follow.

One or more vertices are finished during each iteration in DiskOP. If the successor fragment F of v is unaffected, DiskOP relaxes the outgoing super edges of v in F immediately, and v is the only finished vertex during the iteration; otherwise, DiskOP triggers a PseudoOP process with a closed but un-relaxed vertex v to find CU, and relaxes all the outgoing super edges of the vertices in CU. At the end of the PseudoOP process, all the vertices of CU are finished. Any vertex v that is selected by DiskOP with an affected successor fragment F is the seed triggering a PseudoOP process, denoted as PseudoOP(v, F).

Let T be an outgoing SPT rooted at s in S_Q and T_i be the status of T at the end of the i^{th} iteration that holds all the finished vertices as tree nodes, where $i \in [1, n]$ and n is the number of iterations in DiskOP. Assume that in the $(i+1)^{st}$ iteration, vertex v with affected successor fragment F is selected. Define $UR(v, F)$ as a set of unfinished boundary vertices in F at the time v is selected such that, for any vertex $x \in UR(v, F)$, $SD(v, x, F) \neq +\infty$ and $v \notin UR(v, F)$. Obviously, $e(v, x, F)$ is the outgoing super edge of v in S_Q with a finite weight.

The DiskOP continues the tree computation based on T_i by ignoring the forbidden edge constraint until each vertex of $UR(v, F)$ is closed. Let T_i' be the obtained tree. Intuitively, in order to maximize the number of unfinished boundary vertices to be relaxed with the seed v together in F, we should wait until T_i' contains all the unfinished boundary vertices in F. However, if there exists a boundary vertex z in F such that there is no path available from s in G, to build T_i' is time consuming. Consequently, we choose $UR(v, F)$ as the termination condition to guarantee that only a limited number of boundary vertices are involved in computing T_i'. Let $w_m(UR(v, F))=max(\{SD(v, x, F)|x \in UR(v,$

$F)\})$. Therefore, for any tree node y in T_i', the weight of the tree path from s to y is less than or equal to OD(s, v, G, E_f)+w_m(UR(E_f, F)). According to the partition approach used in [31], we can assume w_m(UR(v, F)) is not large wrt the entire graph G. As a result, obtaining T_i' does not consume too much time.

Since T_i' is built by ignoring the forbidden edge constraint, the relaxation on the outgoing super edges of a boundary vertex inside an affected fragment is *pseudo-relaxation*, and the relaxed super edges are *pseudo-edges*. To categorize the tree nodes in T_i'-T_i, we colour them with *white*, *black* or *grey*. For any vertex $u \in T_i$'-T_i, let p be a path from s to u on T_i'. If any of the edges on p is a pseudo-edge, u is grey. When none of the edges on p is a pseudo-edge, u is either white or black. The successor fragment of the white vertex should be affected, where that of the black vertex is unaffected. We will prove later that the black vertex is finished, the white one is closed but un-relaxed yet and the grey one is open and un-relaxed. Therefore, the grey vertex is *pseudo-closed*, and the white or black vertex is *real-closed*. Figure 4.5 shows the colours of the tree nodes in T_i'-T_i.

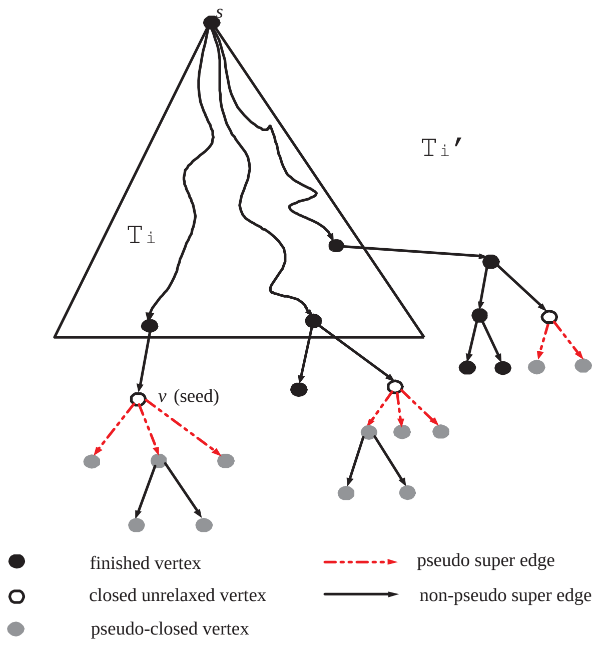


Figure 4.5: Colour of boundary vertex

Denote $CU(v,\ F,\ T_i\text{'-}T_i)$ as a boundary vertex set such that for any vertex $x{\in}CU(v,\ F,\ T_i\text{'-}T_i)$, $x{\in}\ T_i\text{'-}T_i$, and x is a white vertex with F as its successor fragment. The vertices of $CU(v,\ F,\ T_i\text{'-}T_i)$ are the boundary vertices we try to collect. Since all the white vertices are real-closed, their distances are the optimal distances from the source vertex in the graph. Hence, we can compute an OPF rooted at $CU(v,\ F,\ T_i\text{'-}T_i)$ in F with the optimal distances as the value referred to Chapter 1.

Let u be an open boundary vertex in F. If it is a vertex of the OPF, according to the definition of the OPF, it is easy to find an ancestor $a{\in}CU(v,\ F,\ T_i\text{'-}T_i)$ for u such that $OD(s{,}a,\ G,\ E_f){+}OD(a,\ u,\ F,\ E_f){=}min(\{OD(s,\ x,\ G,\ E_f){+}OD(x,\ u,\ F,\ E_f)|x{\in}CU(v,\ F,\ T_i\text{'-}T_i)\})$. As a result, for a super edge $e(x,\ u,\ F)$, where $x{\in}CU(v,\ F{,}T_i\text{'-}T_i)$, it is unnecessary to relax e when u is not in the OPF or x is not the ancestor of u. In other words, in order to finish the relaxation on the vertex set $CU(v,\ F,\ T_i\text{'-}T_i)$, we need to relax only the super edges $e(a,\ u,\ F)$, where a is the ancestor of open vertex u in F. Obviously, the weights of these super edges in S_Q can be retrieved easily from the OPF. The process for the relaxation on vertex set $CU(v,\ F,\ T_i\text{'-}T_i)$ is called *co-relaxation*. The CPU time of building one OPF is almost the same as that of that of building one OPT. Furthermore, the relaxation on$|CU(v,\ F,\ T_i\text{'-}T_i)|$boundary vertices requires reading F only once.

Let ε be the number of the PseudoOP processes triggered by the seeds with F as the successor fragment of the seeds. Therefore, the number of times of reading F is, at most, ε, and our approach computes, at most, ε OPFs in F. In this way, the I/O cost and the CPU time are balanced. Despite the process of building vertex set $CU(v,\ F,\ T_i\text{'-}T_i)$, the saved I/O cost and CPU time is significant.

After the co-relaxation on $CU(v,\ F,\ T_i\text{'-}T_i)$, the finished vertex set in the $(i+1)^{st}$ iteration consists of the black vertex set and $CU(v,\ F,\ T_i\text{'-}T_i)$. The unfinished vertices in T_i' are either the grey vertices or the white vertices which are not in $CU(v,\ F,\ T_i\text{'-}T_i)$. Thus, T_{i+1} can be obtained by removing those unfinished vertices from T_i', and T_{i+1} holds all the finished vertices in S_Q at the end of the PseudoOP process during the $(i+1)^{st}$ iteration of DiskOP. Although our approach employs the same schema as Dijkstra's algorithm, there are some differences between them. Dijkstra's algorithm selects a vertex iteratively, and relaxes its adjacent edges immediately, since the weights of its outgoing edges are available. Also, our approach selects a vertex iteratively. When the successor fragment of the selected vertex is unaffected, DiskOP relaxes the outgoing edges of the vertex at once, but when it is affected, DiskOP calls a PseudoOP process with the vertex as the seed inside the affected successor fragment. Section 4.3.1 describes DiskOP and presents a concrete example, Section 4.3.2 proves the correctness of DiskOP, and Section 4.3.3 analyzes the CPU complexity and the I/O complexity.

4.3.1 Algorithm Description

DiskOP Algorithm

There are ten inputs to DiskOP algorithm: source vertex s, fragment S holding s, destination vertex d, fragment D holding d, the partition of the original entire graph G (also called the fragment database) FDB, DMDB M, sketch graph k, BSDM BM, forbidden edge database E_f where the forbidden edges are grouped by the fragments holding them, and the pivot fragment database PDB. The DiskOP finds a skeleton path from s to d in query super graph S_Q wrt given optimal query $Q(s, d, FDB, E_f)$, and then completes the skeleton path to be an OP to the query. The DiskOP uses the same data structure as DiskSPNN. The only difference is that DiskOP attaches one more *relaxed* attribute to each super node v, indicating whether all the outgoing super edges of v are relaxed or not. The initialization and termination steps of DiskOP are the same as that of NOPT. Since s is relaxed in the step A, DiskOP sets its relaxed attribute as true.

In the iteration step, DiskOP extracts boundary vertex v with the minimum value from priority queue Q iteratively. When v.distance$+\alpha A(bs, d, G)>\gamma A(s, d, G, E_f)$, where bs is the boundary set holding v, DiskOP prunes v and all the open boundary vertices of bs, which is dynamic open boundary vertex pruning, proved in Chapter 3. Otherwise, DiskOP relaxes the outgoing super edges of v. Let F be the successor fragment of v. If F contains a forbidden edge, DiskOP triggers a PseudoOP process with v as the *seed*. The process attempts to find boundary vertex set CU in F, and then relaxes all the outgoing super edges of the vertices of CU inside F by computing an OPF. When F is unaffected, DiskOP conducts a MainThrust on v, relaxing all the outgoing super edges of v in F with its pre-computed DM, as detailed in the Chapter 3. The terminate condition of this step

Algorithm 4.9 DiskOP(s, d, S, D, FDB, M, k, BM, E_f, PDB)

Input: s and d are the source and destination vertices respectively, S and D are the fragments in which s and d are respectively, FDB is the fragment database, M is the DMDB, k is the sketch graph, BM is the BSDM, E_f is the forbidden edge database, PDB is pivot graph database.

Output: an OP from s to d in FDB wrt E_f

```
Step A: Initialization
 1: Build an outgoing OPT $OPT_s$ rooted at $s$ in $S$ wrt $E_f$
 2: Build an incoming OPT $OPT_d$ rooted at $d$ in $D$ wrt $E_f$
 3: $\gamma$=$\gamma$Approx($s$, $S$, $OPT_s$, $d$, $D$, $OPT_d$, $k$, $E_f$, $PDB$)
 4: PruneSketchGraph($s$, $S$, $OPT_s$, $d$, $D$, $OPT_d$, $k$, $BM$, $\gamma$){Prune sketch graph $k$}
 5: Initialize distance array database $daDB$
 6: Initialize priority queue $Q${$Q$ is the U-Heap containing delegates of boundary sets}
 7: $s$.closed=$true$
 8: $s$.distance=0
 9: if $S==D$ then
10:    $d$.distance=$OPT_s$.getDistance($d$)
11:    $d$.predecessor=$s$
12:    $d$.predecessorFrag=$S$
13:    $Q$.update($d$, $d$.distance)
14: end if{relax the super edge $e(s, d, S)$, when $s$ and $d$ are in the same fragment}
15: for each boundary set $b$ in $S$ do
16:    if $k$.contains($b$) then
17:       for each boundary vertex $v \in b$ do
18:          $v$.distance=$OPT_s$.getDistance($v$)
19:          $v$.predecessor=$s$
20:          $v$.predecessorFrag=$S$
21:          $Q$.update($v$, $v$.distance)
22:       end for
23:    end if
24: end for{relax the outgoing super edges of $s$ in $S$}
25: $s$.relaxed=$true$
```

(continued next page)

```
Step B: Iteration
1: while $\neg Q$.empty() do
2:    $v$=$Q$.extractMin()
3:    $v$.closed=$true$
4:    if $v$==$d$ then
5:       goto Step C
6:    end if
7:    find the boundary set $bs$=[$F_1$, $F_2$] in which $v$ locates
8:    if ($v$.distance+$bs$.dstAlpha)>$\gamma$ then
9:       prune $v$ and open vertices in $bs$
10:      remove $bs$ from $k$
11:   end if{boundary vertex pruning, $bs$.dstAlpha is $\alpha A(bs, d, FDB)$ computed in Step
      A}
12:   find successor fragment $F$ of $v${$F$ is either $F_1$ or $F_2$}
13:   if $\neg E_f$.contains($F$) then
14:      $v$.relaxed=$true$
15:      MainThrust($v$, $dist$, $k$, $Q$, $daDB$, $M$, $F$){relax the outgoing super edges of $v$ in $F$}
16:   else
17:      PseudoOP($v$, $dist$, $k$, $Q$, $daDB$, $M$, $F$, $FDB$, $E_f$)
18:   end if
19: end while

Step C: Termination
1: if $d$.closed then
2:    construct the complete OP $p$ from the obtained skeleton path
3: else
4:    $p$=$null$
5: end if
6: return $p$
```

is either the extracted vertex is d, meaning that a skeleton path from s to d is found, or the min-priority queue Q is empty, indicating there is no path from s to d satisfying the given forbidden edge constraint E_f.

Let n be the number of iterations in step B of DiskOP. Denote Q_i as the status of priority queue Q at the end of the i^{th} iteration, where $i \in [1, n]$, and Q_0 is the status at the beginning of the first iteration. Step B, line 2 demonstrates that a vertex is extracted from Q during each loop. Let $\langle \lambda_1, \lambda_2, \ldots, \lambda_n \rangle$ be the extracted vertex sequence, and λ_i be the one extracted in the i^{th} iteration, where $i \in [1, n]$. Denote F_i as the successor fragment of λ_i. Correspondingly, we have a successor fragment sequence $\langle F_1, F_2, \ldots, F_n \rangle$. According to step B, lines 13–18, when F_i does not contain a forbidden edge, DiskOP relaxes the outgoing super edges of i inside F_i in S_Q with the pre-computed DM of F_i; otherwise, trigger a PseudoOP process with i as the seed in F_i.

PseudoOP Process

From step B, lines 13–18 of DiskOP, the PseudoOP process is called only when the successor fragment F of an extracted vertex v contains a forbidden edge. Assume DiskOP triggers PseudoOP in the i^{th} iteration with seed λ_i. Denote the tree status at the end of the j^{th} iteration of PseudoOP as T_{i-1}^{j}, where $j \in [1, m]$, m is the number of iterations in PseudoOP, and T_{i-1}^{0} is the tree status exactly after the initialization step which is the tree with λ_i adding into T_{i-1}. There are two queues; a global queue Q and a local queue psQ in PseudoOP. Q and psQ hold the candidate vertices to be real-closed and pseudo-closed, respectively. Let Q_{i-1}^{j} and psQ_j be the status of Q and psQ, respectively, at the end of the j^{th} ($j \in [1, m]$) iteration of PseudoOP. Q_{i-1}^{0} and psQ_0 are their status at the end of the initialization step of PseudoOP. After the unfinished tree nodes are trimmed from T_{i-1}^{m}, we obtain tree T_i and all of the tree nodes are finished.

According to PseudoOP, step B, line 2, an item with the minimum value of all the items in Q and psQ is extracted iteratively. Let $\langle \mu_1, \mu_2, \ldots, \mu_m \rangle$ be the extracted vertex sequence, and $\langle f_1, f_2, \ldots, f_m \rangle$ be the successor fragment sequence, where μ_j is the vertex extracted in the j^{th} iteration with successor fragment f_j ($j \in [1, m]$). Before describing the algorithm PesudoOP, we introduce several new terms and some symbols. Given graph

Algorithm 4.10 PseudoOP(v, $dist$, k, Q, $daDB$, M, F, FDB, E_f)

Input: v is the seed, $dist$ is the optimal distance of v, k is the sketch graph, Q is the U-Heap containing delegates of boundary sets, $daDB$ is the distance array of boundary sets, M is the DMDB, F is the successor fragment of u, FDB is the fragment database, E_f is the forbidden edge database

Output: none

Ensure: real-close a set of open boundary vertices B in F and do relaxation on vertices in B and v together in F

```
Step A: Initialization
    Initialize pseudo-distance array database $psdaDB$
    Initialize priority queue $psQ${$psQ$ is local queue and $Q$ is global queue}
    PseudoThrust($v$, $dist$, $k$, $daDB$, $psQ$, $psdaDB$, $M$, $F$)
    if $psQ$.empty() then
        return
    end if{no outgoing super edges of $u$ needed to relaxed in $F$}
    Find an un-relaxed boundary vertex set $UR$ in $F$, s.t. for any $x \in UR$, $SD(v, x, F) \neq +\infty$
    and $v \in UR${find un-relaxed outgoing super edges of the $v$ in $F$}
    $CU$=new Set(){a set to hold all white boundary vertices with $F$ as successor fragment}
    $CU$.add($v$)
    $BQ$=new Set(){a set to hold vertices needed to be recovered in $Q$}
(Continued next page)
```

Step B: Iteration

```
 1: while ¬$UR$.empty() do
 2:   ($minv$, $mindist$, $grey$)=ExtractMin($Q$, $psQ$)
 3:   if grey then
 4:     $minv$.pseudoClosed=$true${$minv$ is from $psQ$}
 5:     if $minv$.closed then
 6:       continue{$minv$ has already been real-closed}
 7:     end if
 8:   else
 9:     if ($minv$==$d$)∨($minv$.pseudoClosed) then
10:       $BQ$.add($minv$)
11:       continue
12:     end if{$minv$ has been pesudo-closed or is $d$}
13:     $minv$.closed=$true${$minv$ is real-closed}
14:   end if
15:   find the boundary set $bs$=[$F_1$, $F_2$] to which $minv$ belongs
16:   find successor fragment $F_0$ of $minv${$F_0$ is either $F_1$ or $F_2$}
17:   if $UR$.contains($minv$) then
18:     $UR$.remove($minv$)
19:     if (¬grey)∧($F_0$==$F$) then
20:       $CU$.add($minv$)
21:     end if
22:   end if
23:   if (¬$grey$)∧($F_0$==$D$) then
24:     if $d$.distance>($OPT_d$.getDistance($minv$)+$mindist$) then
25:       $d$.distance=$OPT_d$.getDistance($minv$)+$mindist$
26:       $d$.predecessor=$minv$
27:       $Q$.update($d$, $d$.distance)
28:     end if
29:   end if{relax to $d$}
30:   if (¬$grey$)∧(¬$E_f$.contains($F_0$)) then
31:     $minv$.relaxed=$true${$minv$ is black}
32:     MainThrust($minv$, $mindist$, $k$, $Q$, $daDB$, $M$, $F_0$)
33:   else
34:     PseudoThrust($minv$, $mindist$, $k$, $daDB$, $psQ$, $psdaDB$, $M$, $F_0$)
35:     if (¬grey)∧($F_0$≠$F$) then
36:       $BQ$.add($minv$){$minv$ is white vertex and not in $CU$}
37:     end if
38:   end if
39: end while(Continued next page)
```

```
Step C: Maintenance
 1: for each boundary vertex $b \in BQ$ do
 2:   $Q$.update($b$, $b$.distance)
 3: end for{recover those extracted unfinished vertices in $Q$}
```

```
Step D: Co-relaxation
 1: find open boundary vertex set $OV$ in $F$
 2: OPF($F$, $CU$, $OV$, $E_f$)
 3: for each vertex $b \in OV$ do
 4:   if $OV$.getDistance($b$)$<b$.distance then
 5:     $b$.distance=$OV$.getDistance($b$)
 6:     $b$.predecessor=$OV$.getAncestor ($b$)
 7:     $b$.predecessorFrag=$F$
 8:     $bQ$.decreaseKey($b$, $b$.distance){$bQ$ is the Fib Heap of the boundary set holding $b$}
 9:     $Q$.update($b$, $b$.distance)
10:   end if
11: end for
12: for each boundary vertex $b \in CU$ do
13:   $b$.relaxed=$true$
14: end for{set the relaxed attribute of vertices in $CU$ $true$}
```

Algorithm 4.11 PseudoThrust(u, $dist$, k, $daDB$, psQ, $psdaDB$, M, F_0)

Input: u is the closed boundary vertex, $dist$ is the optimal distance of u, k is the sketch graph, $daDB$ is the distance array of boundary sets, psQ is the local priority queue containing pseudo-relaxed boundary vertices, $psdaDB$ is the pseudo-distance array database containing auxiliary pseudo data of vertices, M is the DM, F_0 is the successor fragment of u.

Output: none

Precondition: F_0 contains forbidden edge

Ensure: u's outgoing super edges in F_0 is pseudo-relaxed

```
1: $m$=$M$.get($F_0$)
2: find adjacent boundary sets $B$ of $bs$ from $k$
3: for each boundary set $b \in B$ do
4:   if $b$.inFragment($F$) then
5:     if $\neg psdaDB$.contain($b$) then
6:       initialize a pseudo distance array of $b$ in $psdaDB$
7:     end if
8:     for each boundary vertex $v \in b$ do
9:       if $\neg v$.pseudoClosed $\wedge$ $\neg v$.closed then
10:        if $v$.distance>($dist$+$m$.get($u$,$v$)) $\wedge$
           $v$.pseudoDist>($dist$+$m$.get($u$,$v$)) then
11:          $v$.pseudoDist=$dist$+$m$.get($u$,$v$)
12:          $v$.pseudoPredecessor=$u$
13:          $v$.pseudoPredeFrag=$F_0$
14:          $bQ$.decreaseKey($v$, $v$.distance) {$bQ$ is the Fib heap of $b$}
15:          $psQ$.update($v$, $v$.distance)
16:        end if
17:      end if
18:    end for{pseudo-relax the outgoing super edges of $u$ in $F_0$ }
19:  end if
20: end for
```

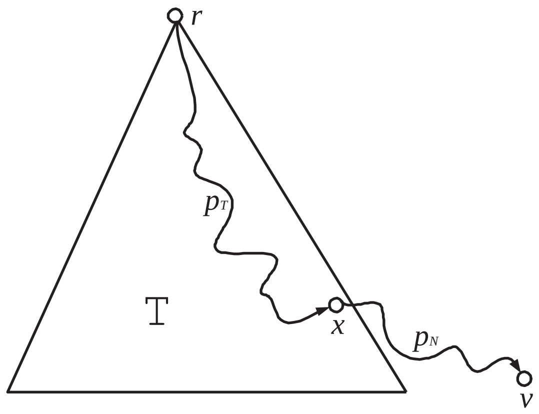


Figure 4.6: Composed path p

$H(V, E, w)$, *reduced edge set* E_d and weight function w', where $E_d \subseteq E$, for each edge $e \subseteq E_d$, decrease its weight $w(e)$ to $w'(e)$, and $0 \leq w'(e) \leq w(e)$ to obtain a *reduced graph graph* H' of H wrt E_d and w'. Denoted H' as $H(E_d, w')$. In other words, H' is the same as H except that the weights of some edges are decreased. Let T be a partial outgoing SPT with root r for both H and H'. Let p be an SP from r to vertex v in H', where v is not on T. Assume vertex x is the last tree node of T on the vertex sequence of path p. Let p_T be the sub-path from r to x on p, and p_N be the sub-path from x to v on p. If p_T is an SP in H' but not a tree path of T from r to x, we can replace p_T with the tree path of T. Hence, any SP p from r to a non-tree node v in H' is regarded as a *composed path* consisting of tree path p_T from r to tree node x in T, and an SP from x to v not via any tree node in H' (Figure 4.6).

Lemma 4.2. Given a graph $H(V, E, w)$ and its reduced graph H' wrt E_d and w', let p be an SP from vertex r to vertex v in H',

(a) if $\forall e \in p \Rightarrow e \notin E_d$, then p is an SP in graph H and $w(p) = SD(r, v, H)$;

(b) if $\exists e \in p$ such that $e \in E_d$, then $w(p) \leq SD(r, v, H)$.

Proof Let p' be any path from r to v in H', then $w(p') \geq w(p)$. Because H is the graph with the weights of a set of edges increased in H', let p'' be the corresponding path of p' in H. Then, $w(p'') \geq w(p')$ and $w(p'') \geq w(p)$. In other words, $w(p)$ is less than or equal to the weight of any path from r to v in H. For case (a), p is valid in H. Therefore p is an SP in H and $w(p)$=$SD(r,\ v,\ H)$; for case b) , p is invalid in H. When p" is the SP in H, then $w(p)\leq SD(r,\ v,\ H)$.□

Define an edge set E^i_r as follows: $E^i_r\subseteq E_Q$, where E_Q is the edge set of the query super graph S_Q. For any edge $e\in E^i_r$, if $head(e)$ is a tree node of T_{i-1}, then $e\in E^i_r$. Therefore, E^i_r consists of all the outgoing super edges of the finished vertices in the previous (i-1) iterations. Let E^i_u be the complement set of E^i_r and E^i_u= E_Q-E^i_r. For any super edge $e(u,\ v,\ F)\in E_Q$, according to the definition of a query super graph, $w(e)$=$OD(u,\ v,\ F,\ E_f)$, which means that if F contains a forbidden edge, $w(e)\geq SD(u,\ v,\ F)$. Otherwise, $w(e)$=SD(u, v, F). Let w' be a weight function on E^i_u such that for each $e(u,\ v,\ F)\in E^i_u$, w'(e)=$SD(u,\ v,\ F)\leq w(e)$. Denote S^i_Q as a reduced graph of S_Q wrt the reduced edge set E^i_u and w'.

Lemma 4.3. Given $1\leq x<y\leq n$, let p be a path from s in T_{x-1}. If p is an SP in S^x_Q, then p is also an SP in S^y_Q.
Proof Because $T_{x-1}\subseteq T_{y-1}$, $E^x_r\subseteq E^y_r$ and $E^x_u\supseteq E^y_u$. Therefore, S^x_Q is the reduced graph of S^y_Q wrt E^x_u-E^y_u and w'. Let e be an edge of p, since p is a path in T_{x-1}, $e\in T_{x-1}$. According to the definition of S^x_Q, $e\in E^x_r$, $e\notin E^x_u$ and $e\notin E^x_u$-E^y_u. Because p is an SP in S^y_Q, according to *Lemma 4.2* (a), p is also an SP in S^y_Q.□

The essence of PseudoOP is to continue the OPT computation based on T_{i-1} in S^i_Q until tree T_{i-1}' is obtained including all the vertices in $UR(\lambda_i,\ F_i)$. For any tree node in T_{i-1}, there exists $S^{i'}_Q(0<i'<i)$ such that the tree path from s is an SP in $S^{i'}_Q$. From *Lemma 4.3*, we can conclude that T_{i-1} is the partial SPT in S^i_Q. Later, it will be proved that T^m_{i-1} is also a partial SPT in S^i_Q. The PseudoOP process builds vertex set $CU(\lambda_i,\ F_i,\ T^m_{i-1}$-$T_{i-1})$, and does a co-relaxation on $CU(\lambda_i,\ F_i,\ T^m_{i-1}$-$T_{i-1})$. Finally, the PseudoOP process removes the unfinished vertices from T^m_{i-1} to create a new tree T_i, and maintains min-priority queue Q such that Q contains all the relaxed data of the nodes in T_i. Then, the DiskOP continues the $(i+1)^{st}$ iteration. Again, the PseudoOP algorithm is another variant of Dijkstra's algorithm. Intuitively, it is assumed that the shortest distance between the two boundary vertices in the same fragment is small. Computing T^m_{i-1} will not cost very much time.

In order to distinguish the real-closed vertices from the pseudo-closed vertices, local min-priority queue *psQ* and local distance array database *psdaDB* are used. The auxiliary data of vertex u in *psdaDB* are u.pseudoClosed, u.pseudoDist, u.pseudoPredeFrag, and u.pseudoPredecessor. Likewise, the auxiliary data of u in *daDB* are u.closed, u.distance, u.predecessorFrag, and u.predecessor. For any vertex u in Q, its potential predecessor u_p must be finished, and the path from the source vertex to u via u_p does not contain any pseudo-edge. For any vertex pu in psQ, there exists at least one pseudo-edge on the path from the source vertex to pu via its pseudoPredecessor. A vertex can have one copy in Q with auxiliary data in *daDB*, and another copy in *psQ* with auxiliary data in *psdaDB*, simultaneously. Let x be a vertex in Q. During the relaxation of super edge $e(y, x, F_y)$ with y.dist+$SD(y, x, F_y)$<x.distance, if y is pseudo-closed or F_y is affected, then another copy of x is inserted into *psQ*. It is possible to extract a vertex twice during the *while* loop of the PseudoOP process. The first extracted copy of the vertex determines if the vertex is real-closed or pseudo-closed, because the PseudoOP process extracts vertices in ascending order wrt their distance attribute.

Define operation ExtractMin(Q, psQ) to extract a vertex with the minimum distance attribute of all the items in Q and *psQ*. Let v_1 and v_2 be the vertex with the minimum distance in Q and *psQ*, respectively. When v_1.distance is equal to v_2.pseudoDist, Extract-Min(Q, psQ) extracts v_1 from Q. It returns *minv*, *mindist*, and a Boolean value *grey*, to represent the extracted vertex, its distance attribute, and whether it is from *psQ* or not, respectively.

PseudoOP extracts a vertex from Q and *psQ* iteratively. If the extracted vertex is from Q and has not been extracted from *psQ* in the previous iterations, it is real-closed. If the vertex is from Q and has been extracted from *psQ* in some previous iteration, it is pseudo-closed. If it is from *psQ* and has not been real-closed yet, it is also pseudo-closed. Otherwise, u is still real-closed. Thus, we can colour the pseudo-closed vertices grey, the real-closed vertices with the affected successor fragment white, and the real-closed vertices with unaffected successor fragment black. PseudoOP conducts a MainThrust on a black vertex; whereas it conducts a PseudoThrust on a white or grey vertex, because a vertex with a white or grey predecessor in T_{i-1}' is grey and pseudo-closed. Both the MainThrust and PseudoThrust relax the outgoing super edges of the extracted vertex with the DMDB.

The MainThrust enqueues the tail vertices of the relaxed super edges into the global queue Q, and updates their auxiliary data in the global distance array database *psdaDB*. However, the PseudoThrust enqueues the tail vertices of the relaxed super edges into local queue *psQ*, and puts their auxiliary data in local distance-array database *psdaDB*. Once a vertex is extracted and processed in the *while* loop of the PseudoOP process, it is either real-closed or pseudo-closed. Consequently, we can add the vertex to tree T_{i-1}. The DiskOP initializes each boundary set of the pruned sketch graph in *daDB*. Boundary set *bs* is initialized in *psdaDB*, if and only if boundary vertex $v \in bs$ is inserted into *psQ* in the PseudoOP process. Because a PseudoOP *while* loop terminates when all the vertices in $UR(\lambda_i,\ F_i)$ are closed or pseudo-closed, only the limited boundary sets are involved in the PseudoOP process. As a result, it is unnecessary to initialize all the boundary sets in *psdaDB*.

The PseudoOP algorithm can easily construct vertex set $CU(\lambda_i,\ F_i,\ T^m_{i-1}\text{-}T_{i-1})$, consisting of the extracted white vertices with successor fragment F_i during the *while* loop. PseudoOP calls for the OPF algorithm to build an OPF rooted at $CU(\lambda_i,\ F_i,\ T^m_{i-1}\text{-}\ T_{i-1})$, and then co-relaxes $CU(\lambda_i,\ F_i,\ T^m_{i-1}\text{-}T_{i-1})$. Therefore, all the vertices of $CU(\lambda_i,\ F_i,\ T^m_{i-1}\text{-}T_{i-1})$ are finished after the co-relaxation. Since, in the *while* loop, the PseudoOP process extracts some vertices from Q, which are not finished in this PseudoOP process. Therefore, we need to recover them in Q. These vertices include a) grey vertices extracted twice, first time from *psQ*, and then from Q; and b) white vertices excluding the ones in set $CU(\lambda_i,\ F_i,\ T^m_{i-1}\text{-}\ T_{i-1})$. Denote $BQ(\lambda_i,\ F_i,\ T^m_{i-1}\text{-}\ T_{i-1})$ as the set consisting of the vertices needed to be recovered in Q. There are four operations on an open boundary vertex set OV:

- OV.setDistance(b, *dist*): sets the distance attribute of $b \in OV$ to *dist*;
- OV.getDistance(b): returns the distance attribute of $b \in OV$;
- OV.setAncestor(b, a): sets vertex a as the ancestor of $b \in OV$;
- OV.getAncestor(b): returns the ancestor of $b \in OV$.

In the initialization step, the PseudoOP process does the following: initializes *psQ* to hold the vertices which are potentially pseudo-closed; initializes *psdaDB* to hold the auxiliary data of the potential pseudo-closed vertices; build boundary vertex set *UR* which is $UR(\lambda_i,\ F_i)$ as defined early in this section; initializes set *CU* which becomes $CU(\lambda_i,\ F_i,$

T^m_{i-1}-T_{i-1}) at the end of the iteration step; initializes set BQ; that is, $BQ(\lambda_i$, F_i, T^m_{i-1}- $T_{i-1})$ at the end of the iteration step; conducts PseudoThrust on v to relax the outgoing super edges of λ_i inside F_i with its out-dated DM.

In the iteration step, the PseudoOP process extracts vertex *minv* with the minimum value among all the elements in *psQ* and Q. When *minv* has been pseudo-closed or is destination d, add it into set BQ and continue the next iteration. If *minv* comes from *psQ* and has not been real-closed yet, it is pseudo-closed, and we do a PseudoThrust on it. Otherwise, *minv* is from Q and is real-closed. Denote F_0 as the successor fragment of *minv*. Perform a MainThrust on *minv* in case F_0 is unaffected, and a PseudoThrust on *minv* in case F_0 is affected. In addition, if *minv* is in *UR*, then remove it from *UR*, and add it to set CU when *minv* is real-closed and F_0 is F_i. In this way, all the vertices of CU are real-closed with F_0 as its successor fragment. When the successor fragment of a real-closed vertex *minv* is *D*, we need to relax the super edge $e($*minv, d, D*$)$. For a real-closed and pseudo-relaxed vertex, if its successor fragment is not F_i, it is placed in the set BQ.

At the end of Step B, all the vertices in set CU obtain the optimal distances from s, and we can co-relax them with the seed inside F_i by computing an OPF. Vertex d is the vertex that is specially treated. Since it is the destination, it requires relaxation on its incoming edges, and its pseudo-distance can be ignored. Hence, we need not take care of d in the PseudoThrust. Even if d is the extracted vertex in some iteration of step B from the global queue Q, since the relaxation on the outgoing super edges of the vertices in CU are not completed yet, the distance of d may not be optimal. Therefore, we must insert d back into Q into the maintenance step.

The maintenance step of the PseudoOP process recovers the vertices in set BQ and returns them to global queue Q. Any vertex x in BQ is one of the following two cases:

- After x has been pseudo-closed, the PseudoOP process extracts it from global queue Q again. Since x is actually open, it must be returned to Q after the iteration step;
- x is real-closed, but is pseudo-relaxed. A PseudoThrust is conducted on it, and the forbidden edges inside its successor fragment are ignored. Thus, x is reinserted in Q to wait for its real-relaxation. The real-relaxation on a vertex is either in the MainThrust process or in the step C co-relaxation of the PseudoOP process. However, if x is a pseudo-relaxed boundary vertex with successor fragment F_i, we

need not recover it in Q, because the next step is the real-relaxation on the outgoing super edges of x in F_i.

In the last step, the PseudoOP algorithm co-relaxes set CU inside successor fragment F_i of the seed λ_i. PseudoOP finds the open boundary vertex set OV in F_i, and then computes OPF OPF_{CU} rooted at CU with the optimal distances from s as the value. For each open boundary vertex y in F, if there exists a path from any vertex of CU to y under the forbidden edge constraint, we can find a vertex $x \in CU$ such that $OD(s, x, G) + OD(x, y, F_i) = min(\{OD(s, v, G) + OD(v, y, F_i) \mid v \in CU\})$, and x is the ancestor of y in OPF_{CU}. As a result, y obtains distance *dist* from its ancestor. If *dist* is less than y.distance, x is the potential predecessor of y. After the co-relaxation on CU, all the vertices of CU are finished.

Optimal Path Forest

The OPF algorithm is also a variant of Dijkstra's algorithm. Dijkstra's algorithm computes an SPT rooted at vertex r with value 0, whereas the OPF algorithm builds an OPF rooted at a set of vertices $CU = \{x_1, x_2, \ldots, x_z\}$ with value $\{d[x_1], d[x_2], \ldots, d[x_z]\}$, respectively. Let g be the given graph. Add a virtual vertex r into g with adjacent edges from r to each vertex x in CU with weight $d[x]$. Compute an outgoing SPT T with root r in g', which is the modified graph g by deleting the given forbidden edges. Then, the OPF rooted at CU with value $\{d[x_1], d[x_2], \ldots, d[x_z]\}$ is obtained by removing r from T.

In the initialization step, the OPF algorithm inserts all the vertices with their values into empty priority queue q. In the iteration step, OPF selects vertex u with a minimum value, and relaxes u's outgoing non-forbidden edges iteratively until q is empty. In this way, we build an OPF consisting of a set of OPTs in g. For each vertex u in g, if there is a path from any vertex of CU to u not via a forbidden edge in g, then u is a vertex of the forest, and u is in the tree rooted at $x_a \in CU(1 \leq a \leq z)$ such that $d[x_a] + OD(x_a, u, g, E_f) = min(\{d_b + OD(x_b, u, g, E_f) \mid 1 \leq b \leq z\})$, x_a is the ancestor of u, and u is the *descendent* of x_a. In the last step, the OPF algorithm finds the ancestors for a set of vertices $Y = \{y_1, y_2, \ldots, y_c\}$ in g.

The PseudoOP algorithm calls the OPF algorithm to compute an OPF rooted at vertex set CU in affected fragment F, and finds the ancestor for each open boundary vertex in F.

During the OPF computation, the auxiliary data of vertex v includes $d[v]$, $p[v]$ and c$[v]$ to denote the potential distance, predecessor, and closed status of v, respectively. After it returns to PseudoOP, the distance of each open vertex obtained from the OPF is compared with its corresponding value in *daDB*. If the former value is less than the latter, then its auxiliary data in *daDB* is updated. In this way, DiskOP finishes the relaxations of the vertices in set *CU*. Algorithm 4.12 provides a description of the OPF algorithm and *Lemma 4.4* and *4.5* prove its correctness.

Let τ be the number of iterations in step B. Denote q_a as the status of q at the end of the a^{th} iteration and exactly before the $(a+1)^{st}$ iteration, where $a \in [1,\ \tau]$, and q_0 is q's status at the beginning of the first iteration. From step B, line 2, we conclude that extracted vertex u_a is the minimum item of q_{a-1}. After extracting u_a from q_{a-1} and relaxing its adjacent non-forbidden outgoing edges, we obtain q_{a+1} according to step B, lines 4–12.

Lemma 4.4. Let $\langle u_1, u_2,\ \ldots,\ u_a,\ \ldots,\ u_\tau \rangle$ be the extracted vertex sequence in the OPF algorithm Step B,τ is the number of iterations, if $1 \le a < b \le \tau$, then $d[u_a] \le d[u_b]$.

Proof *Basis*: when $a{=}1$, as per step A, lines 4–7, all the vertices in set CU are the only elements in q before the first iteration. Thus, $u_1 \in CU$ and $d[u_1] = min(\{d[x] \mid x \in CU\})$. Because u_2 is the minimum element in q_1, either $u_2 \in CU$ or $d[u_2]$ is improved by $d[u_1]$; that is, $d[u_1] < d[u_2]$. Obviously, for both cases, $d[u_1] \le d[u_2]$.

Induction: when $1 < a < \tau$, suppose $d[u_{a-1}] \le d[u_a]$, the objective is $d[u_a] \le d[u_{a+1}]$. Because u_{a+1} is the minimum element in q_a, $d[u_{a+1}]$ is either improved by u_a or remains the same as the moment it is in q_{a-1}. For the former case, as per step B, line 8, $d[u_a] < d[u_{a+1}]$; for the latter case, since u_a is the minimum element in q_{a-1}, $d[u_a] \le d[u_{a+1}]$. As a result, $d[u_a] \le d[u_{a+1}]$.

From the basis and induction, if $1 \le a \le b \le \tau$, then $d[u_a] \le d[u_b]$.□

Lemma 4.4 shows that the extracted vertex sequence is in the ascending order wrt to the distance dimension which is $d[u_1] \le d[u_2] \le \ldots \le d[u_\tau]$. In other words, for any two extracted vertices u_a and u_b, if $d[u_a] \le d[u_b]$, it can be inferred that ua is extracted earlier than u_b, and $1 \le a \le b \le \tau$.

Algorithm 4.12 OPF(F, CU, OV, E_f)

Input: F is a fragment, CU is a real-closed boundary vertex set, OV is an open boundary vertex set, E_f is the forbidden edge database.

Output: the optimal distances from the vertices in CU to those of OV

Precondition: F is an affected fragment, CU consists of all closed un-relaxed boundary vertices in F, OV consists of all open boundary vertices in F.

Step A: Initialization

```
1: $fE$=$E_f$.get($F$){get the forbidden edge set for fragment $F$}
2: Initialize a local auxiliary data map for vertices in $F$
3: Initialize a min-priority queue $q$
4: for each vertex $v \in CU$ do
5:   $d[v]$=$CU$.getDistance($v$){get the optimal distance from $s$ to $v$ as the value of $v$}
6:   $q$.enqueue($v$, $d[v]$)
7: end for
8: $s$.relaxed=$true$
```

Step B: Iteration

```
1: while $\neg q$.empty() do
2:   $u$=$q$.extractMin()
3:   $c[u]$=$true$
4:   for each outgoing edge $e(u,v)$ of $u$ do
5:     if $\neg fE$.contains($e$) then
6:       if $d[v] > (d[u] + w(e))$ then
7:         $p[v]$=$u$
8:         $d[v]$=$d[u]+w(e)$
9:         $q$.enqueue($v$, $d[v]$)
10:      end if
11:    end if
12:  end for{relax edge $e$ if it is not forbidden}
13: end while
```

(Continued next page)

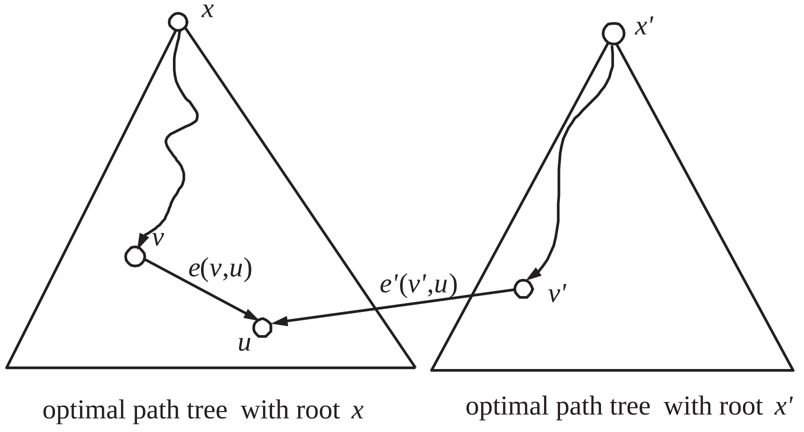
```
Step C: Finding Ancestor
1:  for each vertex $v \in OV$ do
2:      if $v$.closed then
3:          $u$=$v$.predecessor
4:          while $\neg CU$.contains($u$) do
5:              $u$=$p[u]$
6:          end while
7:          $OV$.setAncestor($v$, $u$)
8:          $OV$.setDistance($v$, $d[v]$)
9:      else
10:         $OV$.setAncestor($v$, $null$)
11:         $OV$.setDistance($v$,$+\infty$)
12:     end if{find $v$'s ancestor}
13: end for
```

Lemma 4.5. Given a set of vertices $CU=\{x_1,\ x_2,\ \ldots,\ x_z\}$ with value $\{d[x_1],\ d[x_2],\ \ldots,d[x_z]\}$, respectively, let u be the extracted vertex from priority queue q in the i^{th} iteration of algorithm OPF step B, where $i\in[1,\ \tau]$, then $d[u]=min(\{d[x_b]+OD(x_b,\ u,\ F,\ E_f)\ \mid\ b\in[1,\ z]\})$.

Proof *Basis*: when i=1, from step A, lines 4–7 and step B, line 2, $u\in CU$ and $d[u]=d[u]+OD(u,\ u,\ F)=min(\{d[x_b]\ \mid\ b\in[1,\ z]\})=min(\{d[x_b]+OD(x_b,\ u,\ F,\ E_f)\ \mid\ b\in[1,\ z]\})$.

Induction: when $1<i\leq\tau$, if it is assumed that the induction hypothesis holds for l, where $1\leq l<i$, it can be proved that in the i^{th} iteration, u is extracted from q with $d[u]=min(\{d[x_b]+OD(x_b,\ u,\ F)\ \mid\ b\in[1,\ z]\})$. Let v be the predecessor of u, and v is extracted from q in the l^{th} iteration. In step B, lines 7–8, $l<i$ and $d[v]\leq d$[u]. From the assumption, $d[v]=min(\{d[x_b]\ +OD(x_b,\ v,\ F,\ E_f)\ \mid\ b\in[1,\tau]\})$. Let $x\in CU$ be the ancestor of v such that $d[v]=d[x]+OD(x,\ v,\ F,\ E_f)$.

Suppose there exists vertex v', where edge e'(v', u) is not a forbidden edge in fragment F such that $d[v']+w(e')<d[v]+w(e)=d[u]$. Because $d[v']<d[u]$, v' should be extracted from q earlier than in the i^{th} iteration, say in the l'^{th} iteration ($l'<i$). When $l'<l$, v' is extracted from q earlier than v and in the l'^{th} iteration, from step B, lines



x
x'
v
$e(v,u)$
$e'(v',u)$
v'
u
optimal path tree with root x
optimal path tree with root x'

Figure 4.7: Correctness of the OPF algorithm

5–11, $d[u]$=$d[v']$+$w(e')$. Later in the l^{th} iteration, because $d[v']+w(e')<d[v]+w(e)$, $d[u]$ remains unchanged until the i^{th} iteration. Consequently, $d[u]$=$d[v']$+$w(e')$, which contradicts with our choice of u. When l ¡ l', v is extracted from q earlier than v' and in the l^{th} iteration. From step B, lines 5-11, $d[u]$=$d[v]$+$w(e)$. However in the l'^{th} iteration, because $d[v']+w(e')<d[v]+w(e)$, $d[u]$ is updated with $d[v']+w(e')$, which is a contradiction again. As a result, the supposition is incorrect. For any vertex v' with non-forbidden edge $e'(v', u)$ in F, $d[v]+w(e)\leq d[v']+w(e')$, which is $d[x]+OD(x, v, F)+w(e)\leq d[x']+OD(x, v')+w(e')\leq d[x]+OD(x, v')+w(e')$, where x' is the ancestor of v'. Hence, for any vertex v', $OD(x, v, F)+w(e)\leq OD(x, v')+w(e')$ and $OD(x, u, F)$=$OD(x, v, F)+w(e)$. Now $d[u]$=$d[x]+OD(x, u, F, E_f)$=$min(\{d[x_b]+OD(x_b, u, F) \mid b\in[1, z]\})$ is obtained.

Based on the basis and induction, *Lemma* 4.5 is correct.□

4.3.2 Example of DiskOP

In this section, an example is examined to show how the DiskOP algorithm works. We want to answer an optimal query from s to d in partitioned graph G, where F_3 and F_4 are two affected fragments. During the first nine iterations, DiskOP closes and does a relaxation on v_{18}, v_{19}, v_{20}, v_{23}, v_{17}, v_{21}, v_{22}, v_{15}, and v_{16}, respectively. At this time, the

F_3	v_6	v_7	v_9	v_{10}	v_{11}
v_6	0	8	9	8	12
v_7	8	0	3	8	5
v_9	9	3	0	6	4
v_{10}	8	8	6	0	4
v_{11}	12	5	4	4	0

(a) Distance matrix of F_3

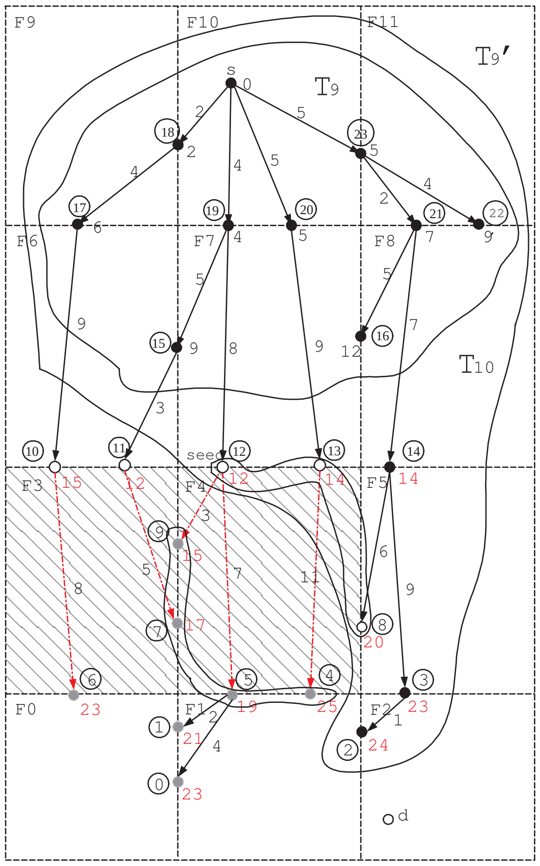
F_4	v_4	v_5	v_7	v_8	v_9	v_{12}	v_{13}
v_4	0	7	8	6	10	14	11
v_5	7	0	3	10	6	7	7
v_7	8	3	0	12	4	6	6
v_8	6	10	12	0	10	8	7
v_9	10	6	4	10	0	3	4
v_{12}	14	7	6	8	3	0	2
v_{13}	11	7	6	7	4	2	0

(b) Distance matrix of F_4

Table 4.1: DMs of affected fragments

partial OPT in S_Q is T_9, and the min-priority queue is Q_9 which holds all the relaxed data of the tree node in T_9.

In the tenth iteration, DiskOP extracted v_{12} from Q_9. Since the successor fragment F_4 of v_{12} contains forbidden edges, v_{12} is the seed to trigger a PseudoOP process in Figure 4.8(a). The OPT computation in the query super graph continues by ignoring the forbidden edge constraint, until the vertices of $UR(v_{12}, F_4)$=$\{v_{13}, v_9, v_7, v_5, v_8, v_4\}$ are added to the tree. First, the PseudoOP process does a PseudoThrust on v_{12}, and enqueues v_9, v_7, v_5, v_8, and v_4 into local queue psQ. The termination condition of the *while* loop in the PseudoOP process is that all the vertices of $UR(v_{12}, F_4)$ are extracted from Q or psQ. Since v_{12} is closed and has an affected successor fragment, v_{12} is white. In the first iteration of PseudoOP *while* loop, v_{11} is extracted from Q with affected successor fragment F_3. Thus, v_{11} is white, and the PseudoOP process conducts a PseudoThrust on it. v_6 is inserted into psQ, and v_7 in psQ is updated. In the remaining iterations of the *while* loop, v_{13}, v_{14}, v_{10}, v_9, v_7, v_5, v_8, v_1, v_3, v6, v_0, v_2, and v_4 are extracted in order. v_8 is extracted twice: the first time is from Q and the second time is from psQ. As a result, it is closed. During this PseudoOP process, v_{12}, v_{11}, v_{13}, v_{10}, and v_8 are white, v_9, v_7, v_5, v_1, v_6, v_0, and v_4 are grey, and v_{14}, v_3, and v_2 are black. It is easy to obtain $CU(v_{12}, F_4, T_9^{14}$- $T_9)$=$\{v_{12}, v_{13}, v_8\}$, because v_{12}, v_{13}, and v_8 are white and their successor fragment is F_4. Since v_{10} and v_{11} are still un-relaxed, the PseudoOP process recovers them in Q. Then the PseudoOP process computes an OPF


(a) PseudoOP trigger by v_{12}

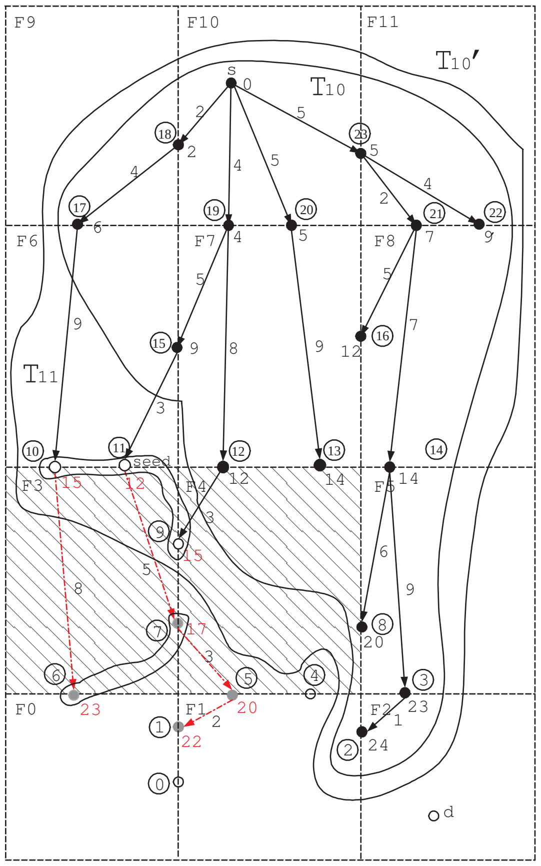

(b) PseudoOP trigger by v_{11}

Figure 4.8: Example of DiskOP algorithm (Part I)

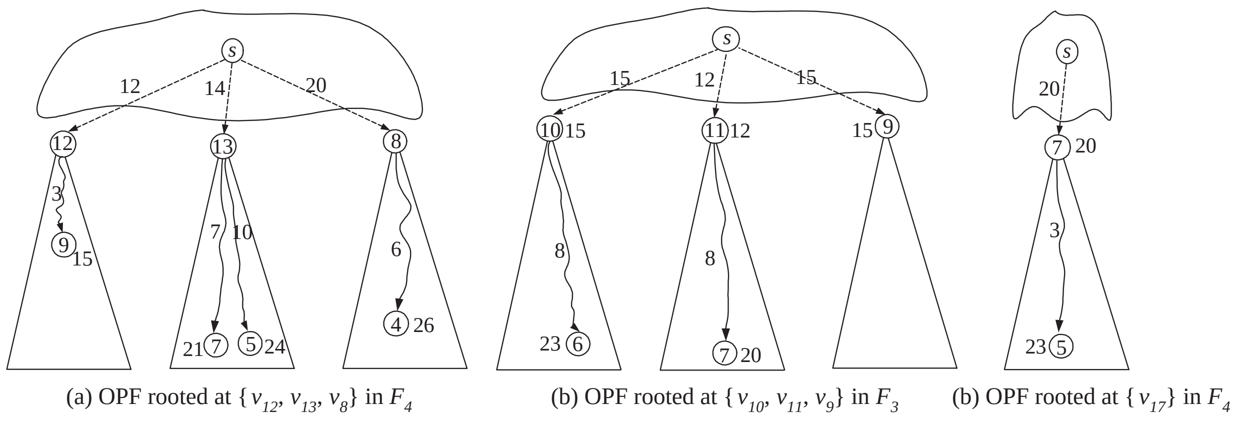

s
12
14
20
12
13
8
3
9 15
7
10
6
21 7
5 24
4 26
(a) OPF rooted at { v12, v13, v8} in F4
15
12
15
10 15
11 12
15 9
8
8
23 6
7 20
(b) OPF rooted at { v10, v11, v9} in F3
20
7 20
3
23 5
(b) OPF rooted at { v17} in F4

Figure 4.9: OPFs of three PseudoOP processes

rooted at $CU(v_{12}, F_4, T_9^{14}\text{-} T_9)$ in F_4 of Figure 4.9(a) and co-relaxes $CU(v_{12}, F_4, T_9^{14}\text{-} T_9)$ so that v_9, v_7, v_5, and v_4 are inserted into Q. In summary, the PseudoOP process finishes all the black vertices and vertices of $CU(v_{12}, F_4, T_9^{14}\text{-} T_9)$, which are added to T_9 to obtain T_{10}. In the previous example, we can see that the optimal distances of v_{16} and v_{12} from s are equal. Here, v_{16} is closed in the ninth iteration of DiskOP. Probably DiskOP chooses to close v_{12} earlier than v_{16} according to the implementation of the min-priority queues. In this case, v_{16} is finished during the PseudoOP process triggered by v_{12}, but the correctness of DiskOP is unaffected. Table 4.1(a)–(b) list the DMs of F_4 and F_3, respectively. The statuses of Q and psQ are in Table 4.2.

In the eleventh iteration of DiskOP, v_{11} is selected. Again, because its successor fragment F_3 is affected, v_{11} is the seed to trigger the PseudoOP process (Figure 4.8(b)). In this process, $UR(v_{11},F_3)=\{v_9, v_{10}, v_7, v_6\}$ and the PseudoOP process selects v_{10}, v_9, v_7, v_5, v_7, v_1, v_6 in sequence in its *while* loop. v_7 is extracted twice: first from psQ then from Q. Therefore, v_7 is pseudo-closed and grey, v_{10}, v_9, v_{11} are white, and v_7, v_5, v_1, v_6 are grey. Obviously, $CU(v_{11}, F_3, T_{10}^{7}\text{-}T_{10})$ is $\{v_{10}, v_9, v_{11}\}$. PseudoOP maintains Q by recovering v_7, and then computes an OPF rooted at $CU(v_{11}, F_3, T_{10}^{7}\text{-}T_{10})$ and co-relaxes $CU(v_{11}, F_3, T_{10}^{7}\text{-} T_{10})$ by updating v_7 and inserting v_6 into Q. Finally, adding the finished vertices v_{10}, v_9, and v_{11} into T_{10}, we have T_{11}. The status of Q and psQ are, in Table 4.3. provided.

In the next iteration of DiskOP, it chooses v_7, triggering a PseudoOP process due to

Q_9	**v_{12}, (v_{19}, F_7,12)**	v_{11}, (v_{15}, F_6,12)	v_{13}, (v_{20}, F_7,14)	v_{14}, (v_7, F_8,14)	v_{10}, (v_{17}, F_6,15)	
Q_9^0	**v_{11}, (v_{15}, F_6,12)**	v_{13}, (v_{20}, F_7,14)	v_{14}, (v_7, F_8,14)	v_{10}, (v_{17}, F_6,15)		
psQ_0	v_9, (v_{12}, F_4,15)	v_7, (v_{12}, F_4,18)	v_5, (v_{12}, F_4,19)	v_8, (v_{12}, F_4,20)	v_4, (v_{12}, F_4,26)	
Q_9^1	**v_{13}, (v_{20}, F_7,14)**	v_{14}, (v_7, F_8,14)	v_{10}, (v_{17}, F_6,15)			
psQ_1	v_9, (v_{12}, F_4,15)	v_7, (v_{11}, F_3,17)	v_5, (v_{12}, F_4,19)	v_8, (v_{12}, F_4,20)	v_6, (v_{11}, F_3,24)	v_4, (v_{12}, F_4,26)
Q_9^2	**v_{14}, (v_7, F_8,14)**	v_{10}, (v_{17}, F_6,15)				
psQ_2	v_9, (v_{12}, F_4,15)	v_7, (v_{11}, F_3,17)	v_5, (v_{12}, F_4,19)	v_8, (v_{12}, F_4,20)	v_6, (v_{11}, F_3,24)	v_4, (v_{13}, F_4,25)
Q_9^3	**v_{10}, (v_{17}, F_6,15)**	v_8, (v_{14}, F_5,20)	v_3, (v_{14}, F_5,23)			
psQ_3	v_9, (v_{12}, F_4,15)	v_7, (v_{11}, F_3,17)	v_5, (v_{12}, F_4,19)	v_8, (v_{12}, F_4,20)	v_6, (v_{11}, F_3,24)	v_4, (v_{11}, F_3,25)
Q_9^4	v_8, (v_{14}, F_5,20)	v_3, (v_{14}, F_5,23)				
psQ_4	**v_9, (v_{12}, F_4,15)**	v_7, (v_{11}, F_3,17)	v_5, (v_{12}, F_4,19)	v_8, (v_{12}, F_4,20)	v_6, (v_{10}, F_3,23)	v_4, (v_{11}, F_3,25)
Q_9^5	v_8, (v_{14}, F_5,20)	v_3, (v_{14}, F_5,23)				
psQ_5	**v_7, (v_{11}, F_3,17)**	v_5, (v_{12}, F_4,19)	v_8, (v_{12}, F_4,20)	v_6, (v_{10}, F_3,23)	v_4, (v_{11}, F_3,25)	
Q_9^6	v_8, (v_{14}, F_5,20)	v_3, (v_{14}, F_5,23)				
psQ_6	**v_5, (v_{12}, F_4,19)**	v_8, (v_{12}, F_4,20)	v_6, (v_{10}, F_3,23)	v_4, (v_{11}, F_3,25)		
Q_9^7	**v_8, (v_{14}, F_5,20)**	v_3, (v_{14}, F_5,23)				
psQ_7	v_8, (v_{12}, F_4,20)	v_1, (v_5, F_1,21)	v_6, (v_{10}, F_3,23)	v_0, (v_5, F_1,23)	v_4, (v_{11}, F_3,25)	
Q_9^8	v_3, (v_{14}, F_5,23)					
psQ_8	**v_8, (v_{12}, F_4,20)**	v_1, (v_5, F_1,21)	v_6, (v_{10}, F_3,23)	v_0, (v_5, F_1,23)	v_4, (v_{11}, F_3,25)	
Q_9^9	v_3, (v_{14}, F_5,23)					
psQ_9	**v_1, (v_5, F_1,21)**	v_6, (v_{10}, F_3,23)	v_0, (v_5, F_1,23)	v_4, (v_{11}, F_3,25)		
Q_9^{10}	**v_3, (v_{14}, F_5,23)**					
psQ_{10}	v_6, (v_{10}, F_3,23)	v_0, (v_5, F_1,23)	v_4, (v_{11}, F_3,25)			
Q_9^{11}	v_2, (v_3, F_5,24)	d, (v_3, F_5,29)				
psQ_{11}	**v_6, (v_{10}, F_3,23)**	v_0, (v_5, F_1,23)	v_4, (v_{11}, F_3,25)			
Q_9^{12}	v_2, (v_3, F_5,24)	d, (v_3, F_5,29)				
psQ_{12}	**v_0, (v_5, F_1,23)**	v_4, (v_{11}, F_3,25)				
Q_9^{13}	**v_2, (v_3, F_5,24)**	d, (v_3, F_5,29)				
psQ_{13}	v_4, (v_{11}, F_3,25)					
Q_9^{14}	d, (v_3, F_5,29)					
psQ_{14}	**v_4, (v_{11}, F_3,25)**					
Q_{10}	v_{11}, (v_{15}, F_6,12)	v_{10}, (v_{17}, F_6,15)	v_9, (v_{12}, F_4,15)	v_7, (v_{13}, F_4,21)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)
	d, (v_3, F_5,29)					

Table 4.2: Status of Q and psQ during the PseudoOP process triggered by v_{12}

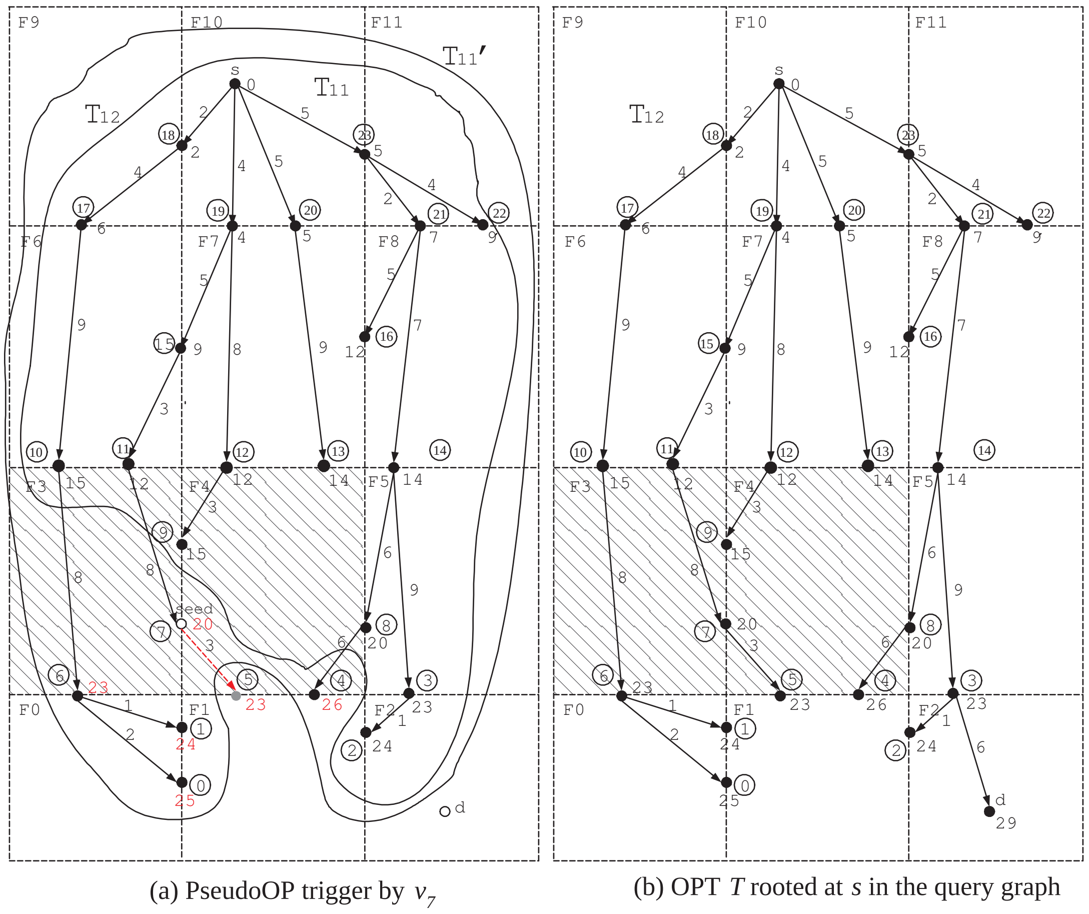

(a) PseudoOP trigger by v_7 (b) OPT T rooted at s in the query graph

Figure 4.10: Example of DiskOP algorithm (Part II)

Q_{10}	**v_{11}, (v_{15}, F_6,12)**	v_{10}, (v_{17}, F_6,15)	v_9, (v_{12}, F_4,15)	v_7, (v_{13}, F_4,21)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)
	d, (v_3, F_5,29)					
Q_{10}^{0}	**v_{10}, (v_{17}, F_6,15)**	v_9, (v_{12}, F_4,15)	v_7, (v_{13}, F_4,21)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)
psQ_0	v_7, (v_{11}, F_3,17)	v_6, (v_{11}, F_3,26)				
Q_{10}^{1}	**v_9, (v_{12}, F_4,15)**	v_7, (v_{13}, F_4,21)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)	
psQ_1	v_7, (v_{11}, F_3,17)	v_6, (v_{10}, F_3,23)				
Q_{10}^{2}	v_7, (v_{13}, F_4,21)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)		
psQ_2	**v_7, (v_{11}, F_3,17)**	v_6, (v_{10}, F_3,23)				
Q_{10}^{3}	v_7, (v_{13}, F_4,21)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)		
psQ_3	**v_5, (v_7, F_4,20)**	v_6, (v_{10}, F_3,23)	v_4, (v_7, F_4,25)			
Q_{10}^{4}	**v_7, (v_{13}, F_4,21)**	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)		
psQ_4	v_1, (v_5, F_1,22)	v_6, (v_{10}, F_3,23)	v_0, (v_5, F_1,24)	v_4, (v_7, F_4,25)		
Q_{10}^{5}	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)			
psQ_5	**v_1, (v_5, F_1,22)**	v_6, (v_{10}, F_3,23)	v_0, (v_5, F_1,24)	v_4, (v_7, F_4,25)		
Q_{10}^{5}	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)			
psQ_5	**v_6, (v_{10}, F_3,23)**	v_0, (v_5, F_1,24)	v_4, (v_7, F_4,25)			
Q_{11}	v_7, (v_{11}, F_3,20)	v_6, (v_{10}, F_3,23)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)	

Table 4.3: Status of Q and psQ during the PseudoOP process triggered by v_{11}

v_7's affected successor fragment F_3 (Figure 4.10(a)). After the PseudoThrust on v_7, v_5 is inserted into psQ. $UR(v_7,\ F_4)$=$\{v_4,\ v_5\}$ and the PseudoOP process selects v_6, v_5, v_5, v_1, v_0, and v_4, in sequence, in its *while* loop. In this PseudoOP process, v_6, v_1, v_0, and v_4 are black, v_7 is white, and v_5 is grey. $CU(v_7,\ F_4)$ is $\{v_7\}$. v_5 is inserted back to Q, since it is extracted twice: first from psQ and then from Q. Then an OPF rooted at $CU(v_7,\ F_4)$ in F_4. Finally, v_5 is updated in Q. During the remaining iterations of DiskOP, a PseudoOP process is not called again. Figure 4.10(b) reflects an OPT rooted at s in query super graph S_Q, and the status of Q and psQ given in Table 4.4.

In this example, the DiskOP algorithm reads F_3 once and F_4 twice, and computes three OPFs. To build T, the NOPT algorithm must compute five OPTs and read F_4 three times, and read F_3 twice, whereas the NDM algorithm computes 5+7=12 SPs and reads F_3 and

Q_{11}	**v_7, (v_{11}, F_3,20)**	v_6, (v_{10}, F_3,23)	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)	
Q_{11}^{0}	**v_6, (v_{10}, F_3,23)**	v_5, (v_{13}, F_4,24)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)		
psQ_0	v_5, (v_7, F_4,23)					
Q_{11}^{1}	v_5, (v_{13}, F_4,24)	v_1,(v_6, F_0,24)	v_0,(v_6, F_0,25)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)	
psQ_1	**v_5, (v_7, F_4,23)**					
Q_{11}^{2}	**v_5, (v_{13}, F_4,24)**	v_1,(v_6, F_0,24)	v_0,(v_6, F_0,25)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)	
psQ_2						
Q_{11}^{3}	**v_1,(v_6, F_0,24)**	v_0,(v_6, F_0,25)	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)		
psQ_3						
Q_{11}^{4}	**v_0,(v_6, F_0,25)**	v_4, (v_8, F_4,26)	d, (v_3, F_5,29)			
psQ_4						
Q_{11}^{5}	**v_4, (v_8, F_4,26)**	d, (v_3, F_5,29)				
psQ_5						
Q_{12}	**v_5, (v_7, F_4,23)**	d, (v_3, F_5,29)				
Q_{13}	**d, (v_3, F_5,29)**					

Table 4.4: Status of Q and psQ during the PseudoOP process triggered by v_7

F_4 once. Obviously, the DiskOP performs much better than NOPT and NDM. Chapter 5 provides more experimental results to prove the efficiency of the DiskOP algorithm.

4.3.3 Correctness of the DiskOP Algorithm

In this section, we prove the correctness of the DiskOP algorithm. As discussed in the last section, DiskOP calls a PseudoOP process when the successor fragment of a selected vertex is affected. If it is assumed in the i^{th} iteration that the PseudoOP process is triggered by seed λ_i, the correctness of PseudoOP is proved according to the following preconditions:
a) T_{i-1} holds all the finished vertices in S_Q, Q_{i-1} contains their relaxed super edges data, and λ_i.distance=$SD(s,\ i,\ S_Q)$;
b) Given 1≤x≤i-1, T_x is a partial SPT in S_Q^x;

c) For any vertex v in S_Q, if $SD(s,\ v,\ S_Q) < SD(s,\ \lambda_i,\ S_Q)$, then v is a tree node of T_{i-1}.

During the *while* loop of PseudoOP, the extracted vertex v in each loop is from either Q or psQ. Consequently, v.dist is denoted as the distance attribute of v when v is extracted, if v is from Q and v.dist=v.distance and v.pred=v.predecessor. Otherwise, it is from psQ, and v.dist=v.pseudoDist and v.pred=v.pseudoPredecessor.

Lemma 4.6. Let μ_j be the vertex extracted in the j^{th} iteration in the PseudoOP process triggered by seed λ_i. Letλ_i be μ_0 and λ_i.distance be μ_0.dist.Suppose $0 \leq j \leq m$, and m is the number of the iterations in PseudoOP for any vertex u in Q^j_{i-1}, μ_0.dist$\leq u$.distance; for any vertex v in psQ_j, μ_j.dist$\leq v$.pseudoDist.
Proof According to step B, line 2, μ_j is the vertex with the minimum distance attribute of all the vertices in Q^{j-1}_{i-1} and psQ_{j-1}.

Basis: when j=0. Before the *while* loop begins, as per PseudoOP step A, line 3, a PseudoThrust is performed on λ_i. psQ_0 holds only the items that are pseudo-relaxed by λ_i.Thus, for any vertex v in psQ_0, μ_0.dist=λ_i.distance$\leq v$.pseudoDist. Since there is no operation on Q in step A, Q^0_{i-1} is the status of Q_{i-1} after extracting λ_i is extracted. Because λ_i is the minimum item in Q_{i-1}, for any vertex u in Q^0_{i-1}, μ_0.dist=λ_i.distance$\leq u$.distance.

Case A: When j=1. If μ_1 is from psQ_0, according to PseudoOP step B, lines 2–16, μ_1 is removed from psQ_0. Since this is the first iteration of PseudoOP *while* loop, μ_1 is pseudo-closed with λ_i as its pseudoPredecessor, and μ_1.closed is *false.* Therefore, PseudoOP conducts a PseudoThrust on μ_1 in step B, line 30. For any vertex v in psQ_1, v.pseudoDist is either improved by μ_1.dist or remains the same as psQ_0, and so λ_i.dist$\leq v$.pseudoDist. Since there is no operation on Q^0_{i-1}, Q^1_{i-1}=Q^0_{i-1}, and for any vertex u in Q^1_{i-1}, μ_1.dist$\leq u$.distance.

Case B: If μ_1 is from Q^0_{i-1}, according to PseudoOP step B, line 2, μ_1 is removed Q^0_{i-1}. From step B, lines 8–14, since this is the first iteration, it is impossible that μ_1 is pseudoClosed. As a result, μ_1.closed is *true.* Then if f_1 is affected, from PseudoOP, step B, line 34, does a PseudoThrust on μ_1. As a result, for any item v in psQ_1, v.pseudoDist either is μ_1.dist+$SD(\mu_1,\ v,\ f_1)$ or remains the same as psQ_0. Consequently, μ_1.dist$\leq v$.pseudoDist. Since Q^1_{i-1} is the status of Q^0_{i-1} after μ_1 is removed, for any item u in Q^1_{i-1}, we have u_1.dist$\leq$ u.distance. If f_1 is unaffected, from PseudoOP, step B, line 32, do a MainThrust on μ_1.

For any vertex u in Q^1_{i-1}, u.distance either is μ_1.dist+$SD(\mu_1,\ v,\ f_1)$ or remains the same as Q^0_{i-1}, and so μ_1.dist$\leq u$.distance. In addition, for any vertex v in psQ_1, since there is no operation on psQ_0, psQ_1=psQ_0, μ_1.dist$\leq v$.pseudoDist. Hence, the lemma holds when j=0, or j=1 for all cases.

Induction: when $2\leq j\leq m$, suppose the induction hypothesis holds for j-1. We wish to show that for any item u in Q^j_{i-1}, μ_j.dist$\leq u$.distance. For any item v in psQ_j, μ_j.dist$\leq$ v.pseudoDist. Case A) μ_j is from psQ_{j-1}. If μ_j has been closed during the previous (j-1) iterations, from PseuoOP, step B, line 6, the PseudoOP process continues the next iteration. Since Q^j_{i-1}=Q^{j-1}_{i-1} and psQ_j is the status of psQ_{j-1}, after removing μ_j, μ_j.dist is no greater than the value of any item in μ_j and psQ_j. Otherwise, μ_j is pseudo-closed, conduct a PseudoThrust on μ_j in step B, line 34, for any vertex v in psQ_j, v.pseudoDist is either improved by μ_j.dist or remains the same as psQ_{j-1}. Thus, μ_j.dist$\leq v$.pseudoDist, and for any vertex u in Q^j_{i-1}, μ_j.dist$\leq u$.distance, because Q^j_{i-1}=Q^{j-1}_{i-1}. Case B) μ_j is from Q^{j-1}_{i-1}. If μ_j has been pseudo-closed in the previous (j-1) iterations, as per step B, lines 9–12, continue the next iteration. In this case, Q^j_{i-1} is the status of Q^{j-1}_{i-1} after removing μ_j and psQ_j=psQ_{j-1}. Obviously, μ_j.dist is no greater than the value of any item in Q^j_{i-1} and psQ_j. Otherwise, μ_j is closed. If f_j is affected, step B, line 34, performs a PseudoThrust on μ_j. Hence, for any vertex v in psQ_j, v.pseudoDist either is μ_j.dist+$SD(\mu_j,\ v,\ f_j)$ or remains the same as psQ_{j-1}, μ_j.dist$\leq$ v.pseudoDist. For any vertex u in Q^j_{i-1}, because Q^j_{i-1} is the status of Q^{j-1}_{i-1} after removing μ_j, μ_j.dist$\leq u$.distance. Otherwise, f_j is unaffected. Step B, line 30 does a MainThrust on μ_j. For any vertex u in Q^j_{i-1}, u.distance either is μ_j.dist+$SD(\mu_j,\ v,\ f_j)$ or remains the same as in Q^{j-1}_{i-1}. Therefore, μ_j.dist$\leq u$.distance. For any vertex v in psQ_j, obviously μ_j.dist v.pseudoDist because psQ_j=psQ_{j-1}. From this analysis, the induction hypothesis also holds for j.

Based on the basis and induction, *Lemma 4.6* is correct □.

According to *Lemma 4.6*, it can easily be inferred that λ_i.distance$\leq\mu_1$.dist$\leq\mu_2$.dist$\leq\ldots\leq$ μ_m.dist. Consequently, let μ_x and μ_y be two extracted vertices in the same PseudoOP process, and $1\leq x,\ y\leq m$. If μ_x.dist<μ_y.dist, then $x<y$; that is, μ_x is extracted earlier than μ_y in the *while* loop of PseudoOP. Based on precondition a), and b), and *Lemma 4.3*, there is *property* 1: for any vertex $v\in T_{i-1}$,$SD(s,\ v,\ S_Q)$=$SD(s,\ v,\ Q^{i-1}_Q)$=$SD(s,\ v,\ Q^i_Q)$.

Theorem 4.1. λ_i.distance=$SD(s,\ \lambda_i,\ S_Q)$=$SD(s,\ \lambda_i,\ S_Q^i)$.

Proof As per precondition a), λ_i.distance=$SD(s,\ \lambda_i,\ S_Q)$. Precondition c) shows that for any vertex $v \notin T_{i-1}$, $SD(s,\ v,\ S_Q) \geq SD(s,\ \lambda_i,\ S_Q)$, and the predecessor of v must be a tree node in T_{i-1}. Furthermore, the outgoing edges of any tree nodes of T_{i-1} are not in the reduced edge set of S_Q^i. From *property* 1, it is inferred that $SD(s,\ v,\ S_Q^i) \geq SD(s,\ \lambda_i,\ S_Q)$. According to DiskOP, step B, line 2, λ_i is the vertex with the minimum distance attribute in Q_{i-1}. Therefore, λ_i.distance=$SD(s,\ \lambda_i,\ S_Q)$=$SD(s,\ \lambda_i,\ S_Q^i)$.□

Theorem 4.2. Given $1 \leq j \leq m$, for extracted vertex μ_j during the *while* loop of the PseudoOP process in the i^{th} iteration of DiskOP($1<i<n$), μ_j.dist=$SD(s,\ \mu_j,\ S_Q^i)$ and T_{i-1}^j=$T_i \cup$ $\mu_0,\ \mu_1,\ \ldots,\ \mu_j\}$.

Proof From *theorem 4.1*, λ_i.distance=$SD(s,\ \lambda_i,\ S_Q)$=$SD(s,\ \lambda_i,\ S_Q^i)$. PseudoOP, step A, line 3, does a PseudoThrust on λ_i, T_{i-1}^0=$T_i \cup \{\ \lambda_i\}$, and for any vertex v in psQ_0, v.pred=λ_j.

Basis: when j=1, μ_i is the minimum item in Q_{i-1}^0 and psQ_0. According to precondition a) and the PseudoOP process, step A, line 3, the predecessor of μ_1 is either λ_i or a tree node of T_{i-1}. Let it be a vertex v with super edge $e(v,\ \mu_1,\ F)$ in S_Q^i, and then μ_1.dist= $SD(s,\ v,\ Q_Q^i)$+$w(e)$, where $w(e)$ is the weight of e in S_Q^i. Suppose there exists vertex v' with super edge e'(v', μ_1, F') in S_Q^i such that $SD(s,\ v',\ S_Q^i)+w(e') < SD(s,\ v,\ S_Q^i)+w(e)$. If $v' \in T_{i-1}^0$, μ_1.pred should be v' instead of v. This is due to the fact that the outgoing super edges of any tree node in T_{i-1}^0 have been relaxed before μ_1 is extracted. If $v \notin T_{i-1}^0$, $SD(s,\ v',\ S_Q^i) \geq SD(s,\ i,\ S_Q)$. Therefore, the assumption is wrong, and μ_1.dist= $SD(s,\ \mu_1,\ S_Q^i)$ and T_{i-1}^1=$T_i \cup \{\ \mu_0,\ \mu_1\}$.

Induction: when $1<j \leq m$, suppose the induction hypothesis holds for j-1, to show that μ_j.dist=$SD(s,\ \mu_j,\ S_Q^i)$. Let v be the predecessor of μ_j, when it is extracted in the j^{th} iteration. μ_j.dist=v.dist+$w(e)$, where e is super edge $e(v,\ \mu_j,\ F)$ in S_Q^i. Assume there is a vertex v' with super edge e'(v', μ_j, F') in S_Q^i such that v'.dist+$w(e')$ <v.dist+$w(e)$. From *Lemma 4.6*, v' is either extracted in the j'^{th} iteration ($j'<j$), or it is a tree node of T_{i-1}^0. For the former case, in the j'^{th} iteration, update either μ_j.distance in Q_{i-1}^j or μ_j.pseudoDist in psQ_j to be v'.dist+$w(e')$ and remain until μ_j is extracted, which contradicts μ_j.dist. For the latter case, when $v' \in T_{i-1}^0$, then either μ_j.distance in Q_{i-1}^0 or μ_j.pseudoDist in psQ_0 is

v'.dist+$w(e$') and is unchanged until extracted, which is a contradiction again.Therefore, for any vertex v' with super edge e'(v', μ_j, F') in S^i_Q, μ_j.dist=v.dist+$w(e)\leq v$'.dist+$w(e$'). From the hypothesis and *property1*, v.dist=$SD(s$, v, $S^i_Q)$. Therefore, μ_j.dist=$SD(s$, v, $S^i_Q)+w(e)=SD(s$, μ_j, $S^i_Q)$.

Based on the basis and induction, *Theorem 4.2* is correct.□

During an specific PseudoOP process, local priority queue *psQ* and local database *psdaDB* are initialized to hold the vertices that may be pseudo-closed, and their auxiliary data. For vertex v, there may be two auxiliary data: one is in global Q and *daDB*, whereas the other is in local *psQ* and *psdaDB*. From the PseudoThrust lines 9–10, the pseudo-auxiliary data of a vertex v is updated only when the data is neither pseudo-closed nor closed, and both its distance and pseudo-distance are greater than the new distance. In this way, we still keep and update the vertex's auxiliary data from the MainThrust, and easily determine if a vertex is closed or pseudo-closed by the queue which it is from. The pseudo-closed vertices are still open, because their optimal distances have not been found yet. Since all the data pseudo-relaxation is in *psQ* and *psdaDB*, when the process PseudoOP ends, the data does not exist. When vertex v has been pseudo-closed and is extracted again from Q later, step B, line 8, records it in set BQ and step C, lined 1–3 returns it to Q. In this way, we keep v in Q and its pseudo-closed status will not affect its search for optimal distance.

Theorem 4.3. Given $1\leq j\leq m$, let μ_j be an extracted vertex during the *while* loop of PseudoOP in the i^{th} iteration of DiskOP ($1<i<n$). If μ_j is either white or black, it is closed and μ_j.distance=$SD(s$, μ_j, $Si_Q)=SD(s$, psQ_j, $S_Q)$; otherwise μ_j is grey, μ_j.distance=$SD(s$, μ_j, $S^i_Q)\leq SD(s$, μ_j, $S_Q)$.

Proof Let p be a tree path from s to μ_j in T^m_{i-1}. According to *theorem 4.2*, $w(p)$=$SD(s$, μ_j, $S^i_Q)$. As per the definition of a black or white vertex, if μ_j is either white or black, there is no edge e on p such that $e\in T^m_{i-1}$-T_{i-1}, and e lies in an affected fragment. As a result, although e is in the reduced edge set of S^i_Q, its weight is the same as that in S_Q. Hence, from *Lemma 4.2*(a), $SD(s$, μ_j, $S^i_Q)$= $SD(s$, μ_j, $S_Q)$ when μ_j is either white or black, and is closed. If μ_j is grey, according to the definition of a grey vertex, there exists an edge e on p such that $e{\in}T^m_{i-1}$-T_{i-1}, and e lies in an affected fragment. As a result, the weight of e is decreased. As per *Lemma 4.2*(b), $w(p){=}SD(s,\ \mu_j,\ S^i_Q){\leq}SD(s,\ \mu_j,\ S_Q)$. □

Theorem 4.4. The outgoing super edges of the vertices in set $CU(\lambda_i,\ F_i,\ T^m_{i-1}$-$T_{i-1})$ are finished in the co-Relaxation step of the PseuoOP process triggered by λ_j.

Proof As per PseudoOP, step B, line 20, F_i is the successor fragment of any vertex x in $CU(i,\ F_i,\ T^m_{i-1}$-$T_{i-1})$, and x is white. Hence, we need to relax only the outgoing super edges of x inside F_i. In addition, it is unnecessary to relax any super edge $e(x,\ v,\ F_i)$, where v is a closed boundary vertex in F_i, because $SD(s,\ v,\ S_Q)$ is obtained. According to *Lemma 4.5*, we can obtain ancestor $a{\in}CU(i,\ F_i,\ T^m_{i-1}$-$T_{i-1})$ in an OPF. The OPF is rooted at $CU(\lambda_j,\ F_i,\ T^m_{i-1}$-$T_{i-1})$ inside F_i for each open boundary vertex y in F_i such that $SD(s,\ a,\ S_Q){+}OD(\text{a},\ y,\ F_i){\leq}SD(s,\ x,\ S_Q){+}OD(x,\ y,\ F_i)$, where $x{\in}CU(\lambda_j,\ F_i,\ T^m_{i-1}$-$T_{i-1})$. As a result, it is unnecessary to relax $e(x,\ v,\ F_i)$, where x is not a. In other words, to perform a relaxation on $CU(\lambda_j,\ F_i,\ T^m_{i-1}$-$T_{i-1})$, we only need to relax any super edge $e(a,\ y,\ F_i)$, where a is the ancestor of y in the OPF, $a{\in}CU(\lambda_j,\ F_i,\ T^m_{i-1}$-$T_{i-1})$ and y is an open boundary vertex in F_i. The PseudoOP co-Relaxation step, the lines 1–2 computes the OPF and lines 3–11 relaxes the previous super edges. Therefore, all the vertices in $CU(\lambda_j,\ F_i,\ T^m_{i-1}$-$T_{i-1})$ are relaxed. From *theorem 4.3*, $CU(\lambda_j,\ F_i,\ T^m_{i-1}$-$T_{i-1})$ is a finished vertex set.□

Theorem 4.5. Given $1{\leq}j{\leq}m$, let μ_j be an extracted vertex during the *while* loop of the PseudoOP process in the i^{th} iteration of DiskOP($1{<}i{<}n$). If μ_j is black, it is a finished vertex.

Proof According to *theorem 4.3* and the precondition a), at the end of the j^{th} iteration, during the *while* loop of the PseudoOP process, a closed vertex v is either a tree node of T_{i-1} or $\mu_{j'}$ ($0{\leq}j'{\leq}$j). Therefore, to conduct a relaxation on μ_j is to relax any super edge $e(\mu_j,\ u,\ f_j)$, where u is open. Because μ_j is black, f_j is an unaffected fragment, and $w(e){=}SD(\mu_j,\ u,\ f_j)$ can be read directly from the DM of f_j. PseudoOP, step B, line 32, does a MainThrust on μ_j to relax its outgoing super edges. Consequently, the black vertices are finished.□

Theorem 4.6. After the PseudoOP process is triggered by λ_j, T_i=T_{i-1} $\cup CU(i, F_i, T^m_{i-1}$-$T_{i-1})\cup$ BLACK, where BLACK is the black vertex set obtained in the PseudoOP process. Q_i holds all the relaxed super edges information of all the tree vertices in T_i. For any vertex v in S_Q, if $SD(s, v, S_Q)<\lambda_{i+1}$.distance, then v is a tree node of T_i.λ_{i+1}.distance=$SD(s, \lambda_{i+1}, S_Q)$.

Proof The PseudoOP process triggered by λ_i finishes the black vertices and vertices of $CU(\lambda_i, F_i, T^m_{i-1}$-$T_{i-1})$ which are added to T_{i-1} to be T_i. Regarding the closed un-relaxed vertices or the pseudo-closed vertices which are extracted from Q, PseudoOP, step B, lines 10, 36 and step C, lines 1–3, recovers the vertices for Q. Furthermore, since PseudoOP does a MainThrust on the finished vertices in the i^{th} iteration of DiskOP, Q_i holds all the relaxed super edges information of all the tree nodes in T_i. Assume there is a vertex v' in S_Q such that $SD(s, v', S_Q)<\lambda_{i+1}$.distance and $v'\notin T_i$. According to DiskOP, line 2, for any vertex x in Q_i, λ_{i+1}.distance$\leq x$.distance. From $v'\notin T_i$, $v'\notin T_{i-1}\subseteq T_i$, as per precondition c) and *Lemma 4.6*, $SD(s, \lambda_i, S_Q)\leq SD(s, v', S_Q)<\lambda_{i+1}$.distance. Therefore, the predecessor of v' is a tree node of T_i. Because Q_i holds all the relaxed super edges information of all the tree nodes in T_i, v' is in Q_i, which contradicts the extraction of λ_{i+1} in the $(i+1)^{st}$ iteration of DiskOP. Therefore, the assumption is wrong. For any vertex v in S_Q, if $SD(s, v, S_Q)<\lambda_{i+1}$.distance, v is a tree node of T_i. From *theorem 4.1*, λ_{i+1}.distance=$SD(s, \lambda_{i+1}, S_Q)$=$SD(s, \lambda_{i+1}, S_Q^{i+1})$.□

Lemma 4.7. Let λ_i be the vertex extracted in the i^{th} iteration of the *while* loop in the DiskOP iteration step. Suppose $0\leq i\leq n$, for any item u in Q_i, λ_i.dist$\leq u$.distance, where λ_0 is source vertex s.

Proof *Basis*: when i=0, step A, line 1 relaxes the outgoing super edges of s with OPT_s inside the source fragment. Therefore, at the end of step A, Q_0 holds the boundary vertices in S, and there exist paths from s to these boundary vertices in S. Hence, for any item u in Q_0, u.distance=$OD(s, u, S)\geq s$.distance=0.

When i=1, λ_1 is the minimum item in Q_0. If F_1 is an unaffected fragment, step B, line 15 conducts a MainThrust on λ_1 in F_1. For any item u in Q_1, u remains the same as Q_0, u.distance$\geq\lambda_1$.distance, or u.distance= λ_1.distance+$SD(\lambda_1, u, F_1)\geq\lambda_1$.distance during the previous MainThrust. If F_1 is an affected fragment, the PseudoOP process is triggered.

According to *Lemma 4.6*, for any item u in Q_1, u.distance$\geq\lambda_1$.distance. As a result, *Lemma 4.7* holds when i=1.

Induction: when 1$\leq i\leq$n, suppose the induction hypothesis holds for i -1. The objective is to show that for any item u in Q_i, λ_i.dist$\leq u$.distance. λ_i is the minimum item in Q_{i-1} according to step B, line 2. If F_i is an unaffected fragment, step B, line 15 performs a MainThrust on λ_i in F_i. For any item u in Q_i, u is either remains the same as Q_i. u.distance$\geq\lambda_i$.distance, or u.distance=λ_i.distance+$SD(\lambda_i,\ u,\ F_i)\geq i$.distance during the previous MainThrust. If F_i is an affected fragment, a process PseudoOP is triggered. According to *Lemma 4.6*, for any item u in Q_i, u.distance$\geq\lambda_i$.distance. Therefore, for any item u in Q_i, λ_i.dist$\leq u$.distance.

Based on the basis and induction, *Lemma 4.7* is correct□.

Corollary 4.1. Let u be a vertex finished in the i'^{th} iteration, where $i\leq i$'. Then, λ_i.distance$\leq u$.distance.

Proof Immediately from *Lemma 4.6* and *4.7*, it can be inferred that λ_i.distance u.distance.□

Theorem 4.7. Let λ_i be the vertex extracted in the i^{th} iteration of the *while* loop in the DiskOP iteration step. Suppose 1$\leq i\leq n$, λ_i.distance=$SD(s,\ \lambda_i,\ S_Q^i)$. If F_i is affected, let x be a finished vertex during the PseudoOP process triggered in i^{th} iteration, and x.distance=$SD(s,\ x,\ S_Q)$=$SD(s,\ \lambda_i,\ S_Q^i)$.

Proof *Basis*: when i=0, step A closes and conducts a relaxation on s and s.distance=$SD(s,\ s,\ S_I)$=0. When i=1, λ_i is the minimum item in Q_0, which is λ_1.distance=$min(\{OD(s,\ u,\ S)|u$ is a boundary vertex of S, if S==D, u can be $d\})$. According to *Corollary 4.1*, there is no other vertex v in S$_Q$ such that v.distance<λ_1.distance. As a result, λ_1.distance=$SD(s,\ \lambda_1,\ S_Q)$=$SD(s,\ \lambda_1,\ S_Q^1)$. If F_1 is unaffected, according to step B, line 15, λ_1 is the only vertex finished in this iteration. Otherwise, because all the preconditions are satisfied, from *theorem 4.6*, x.distance=$SD(s,\ x,\ S_Q)$=$SD(s,\ x,\ S_Q^1)$. Therefore, the preconditions are satisfied at the end of the first iteration.

Induction: when 1$\leq i\leq n$, suppose the induction hypothesis holds for i-1, The desire is to show that λ_1.distance=$SD(s,\ \lambda_1,\ S_Q)$=$SD(s,\ \lambda_1,\ S_Q^i)$. If F_i is affected, let x be a finished vertex the PseudoOP process triggered in the i^{th} iteration, and x.distance= $SD(s$,

λ_1, S_Q)=SD(s, λ_1, S_Q^i). All the preconditions are still valid at the end of the i^{th} iteration. Since λ_i is the vertex with the minimum distance attribute in Q_i, and all the preconditions are valid at the end of the $(i-1)^{st}$ iteration, according to *theorem 4.1*, λ_i.distance=SD(s, λ_i, S_Q)=SD(s, λ_i, S_Q^i). If F_i is unaffected, DiskOP, step B, line 15 does a MainThrust on λ_i, which is finished in this iteration, and T_i=$T_{i-1}\cup\{\lambda_i\}$. In this case, all the preconditions are valid at the end of the i^{th} iteration. If F_i is affected, the PseudoOP process is triggered by λ_i in DiskOP, step B, line 17. Since the preconditions are valid at the end of the $(i-1)^{st}$ iteration from *theorem 4.6*, x.distance=SD(s, x, S_Q)=SD(s, x, S_Q^i). Therefore, the preconditions are satisfied at the end of the i^{th} iteration.

Based on the basis and induction, *theorem 4.7* is correct.□

4.3.4 Complexity Analysis

The complexity of the disk-based OP algorithm includes the CPU complexity and the I/O complexity. The CPU complexity is principally from the update of the main memory data structure, the OPF computations in the affected fragments, the direct relaxation in the *while* loop of DiskOP, and the relaxation during the PseudoOP processes. The I/O complexity consists of I/O of fragment database and DMDB. The sizes of the other disk-based data structures such as distance array database and pivot fragment database are trivial, compared with the previous databases. The size of the pivot fragment database in our experiments is less than 7% of fragment database, which is revisited in Chapter 5. We can ignore the pivot fragments by assuming that they are loaded in the main memory. Since different buffer management schemes result different I/O performance, we assume there is no buffer. Every access on a disk-based data structure leads to an I/O operation.

There are three steps in DiskOP: initialization, iteration, and termination, which are analyzed separately. The initialization steps complete the two OPTs computations inside the source and destination fragments, prunes a sketch graph, initializes the data structures, and performs a relaxation on the source vertex in the query super graph. The iteration step computes a skeleton path in the query graph. The termination step fills each super edge of the skeleton path with the actual path and returns the complete OP. The sum of the three steps is the complexity of DiskOP.

Denote n and m as the average number of vertices and edges in a fragment, respectively. Let b be the average number of boundary vertices in a fragment, s be the number of boundary sets in the whole graph, and z be the average number of super edges of the skeleton path. Assume that accessing a disk-based data structure once results in one B-bytes I/O operation. Let B_1, B_2 be the average I/O bytes by reading one fragment and one DM, respectively.

Since an OPT inside a fragment is actually the SPT in the modified fragment by deleting the forbidden edges, the CPU complexity of the OPT computation is the same as that of Dijkstra's algorithm which is $O(n\lg n+m)$. According to the analysis in Section 4.3.1, the CPU complexity of an OPF computation and that of an OPT are the same: $O(n\lg n+m)$.

In the following, the complexities of DiskOP's initialization step, termination step, and iteration step are analyzed respectively. The sum of the complexity is DiskOP's complexity.

Complexity of Initialization and Termination Step

In the initialization step, lines 1–2 build two OPTs in the source and destination fragment, respectively, whose CPU complexity is $O(n\lg n+m)$. Also, the two reading operations on fragment database with I/O bytes is $O(B_1)$. Line 3 calls algorithm 4.5 to compute the γ-approximation of an OP; line 4 calls algorithm 3.3 to prune the sketch graph. The algorithm traverses the sketch graph with BFS, and computes the α-approximation for each visited sketch node.

Algorithm 4.5 is a variant of Dijkstra's algorithm on the sketch graph. Since both the sketch graph and pivot fragment database are loaded into the main memory, to calculate a γ-approximation does not lead to any I/O activity. The iteration step of algorithm 4.5 selects close boundary set bs and relax it. When a boundary set bs is closed with an affected successor fragment F, step B, lines 15-20 compute an OPT inside the pivot fragment of F to update the weights of incident sketch edges of bs in F. When F does not contain a forbidden edge, the weights of the sketch edges inside F remain unchanged. Then, step B, lines 21–27 relax the incident sketch edges of bs inside F. Therefore, we regard algorithm 4.5 as Dijsktra's algorithm with a real-time edge weights re-computation. According to the definition of a sketch graph, the number of vertices in the sketch graph is the number of boundary sets in the entire graph. Let k_m be the number of sketch edges in the sketch graph and k_T be the number of computed OPTs in the iteration step. Assume p_n and p_m are the average number of vertices and edges in a pivot fragment respectively. Then, the CPU complexity on the re-computation of the sketch edges' weights is $O(k_T[p_n\lg p_n+p_m])$. As a result, the CPU complexity of algorithm 4.5 is $O(k_T[p_n\lg p_n+p_m]+s\lg s+k_m)$.

Algorithm 3.3 is a BFS algorithm with some α-approximations. If it is assuming that the boundary set DM is in the main memory, algorithm 3.3 has no I/O activities. The CPU complexity of lines 1–5 and lines 22–26 is $O(k_m)$. The α-approximation operation (lines 10-11) takes $O(1)$ time. Thus, the total time of the α-approximation operations are $O(\mathrm{s})$. In addition, the breadth first search in the *while* loop (lines 6–21) takes $O(s+k_m)$ time. Consequently, the CPU time of algorithm 3.3 is $O(s+k_m)$.

The DiskOP initialization step, lines 5–6 initializes distance array database and global priority queue Q. Each element in the distance array database holds the distance array of a boundary set, and each distance array has a Fibonacci heap for the vertices in each boundary set and attached attributes of the boundary vertices. Q is a U-Heap holding the delegate vertices of all the boundary sets. Therefore, the CPU time for lines 5–6 is $O(b\times s+s)$, and the I/O complexity is ignored. Lines 7–24 do a relaxation on the source vertex which is trivial again, compared to the iteration step. Lines 7–14 take $O(1)$ time. Line 21 is an update operation on Q and executes $O(b)$ times in the loop from lines 15 to 24. From the definition of the update operation on Q in Chapter 3, it consists of one update operation on the U-Heap and one decreaseKey operation on the Fibonacci heap. The U-Heap updates the value of a leaf vertex each time. In the worst case, all the ancestors of the leaf vertex are updated. Consequently, the U-Heap takes $O(\lg s)$. Since a decreaseKey operation takes $O(1)$ amortized time [6], the CPU complexity of lines 7–24 is $O(b\lg s)$. Therefore, the total CPU complexity in the initialization step is $O(n\lg n+m+b\times s+b+k_T[p_n\lg p_n+p_m]+s\lg s+k_m)=O(n\lg n+m+b\times s+k_T[p_n\lg p_n+p_m]+s\lg s+k_m)$, and the I/O bytes is $O(B_1)$.

In the termination step, DiskOP computes z OPTs. Therefore, its CPU complexity is $O(z\times n\lg n+z\times m)$ and I/O bytes are $O(z\times B_1)$.

Complexity of Iteration Step

The kernel of DiskOP algorithm is the iteration step. A boundary vertex v is selected to close during each iteration. Let F be the successor fragment of v. If F is affected, DiskOP calls for a PseudoOP process in step B, line 17. Otherwise, F is unaffected. The DiskOP does a relaxation on v with the DM of F. Therefore, we can divide the iteration step into affected or unaffected successor fragments.

Unaffected Successor Fragments

When F is unaffected, line 2 does an extractMin operation on Q. Lines 3-11 set the status of the extracted boundary vertex v, and dynamically prune the open boundary vertices. From the definition of extractMin operation on Q in Chapter 3, one update operation on a U-heap and one extractMin operation in a Fibonacci heap are included. Since the amortized time of an extractMin operation in a Fibonacci heap is O(lg$b$)[6], line 2 takes $O(\lg s{+}\lg b)$ CPU time. Lines 3–7 and 12 take $O(1)$ time obviously. Lines 8–11 take $O(b)$ time to finish the pruning of the open boundary vertices. Lines 13–16 relax the closed vertex v by calling for a MainThrust process which is algorithm 3.5. A MainThrust process relaxes all the adjacent super edges of v inside F with an update operation on Q. According to the analysis in the last section, it takes $O(b\lg s{+}b)$ CPU time. Therefore, the total CPU time of one loop wrt the unaffected successor fragment is $O(\lg s{+}\lg b{+}b{+}1{+}b\lg s{+}b){=}O(b\lg s)$. Let the total number of the loops wrt an unaffected successor fragment be b_u. Hence, the total CPU complexity is $O(b_u{\times}b\lg s)$.

The main I/O activity is to read the DM of F into the MainThrust, line 1. Therefore, the total I/O bytes are $O(b_uB_2)$.

Affected Successor Fragment

When F is affected, DiskOP still runs lines 2–12 to extract a boundary vertex and to dynamically prune the boundary vertices first. Then, the algorithm calls a PseudoOP process in line 15. From the previous analysis, lines 2–12 take $O(\lg s{+}b)$ time.

There are global data structures and local data structures in DiskOP. Global structures are valid throughout the whole algorithm, and are initialized in the initialization step. Local data structures are created only during a PseudoOP process, consisting of a temporary U-

Heap containing the delegate of the boundary sets involved in the pseudo-relaxation, a Fibonacci heap for the vertices in each involved boundary set, and a binary heap for the vertices inside F. As a result, we have to consider both the global and local data structures in the PseudoOP process.

Algorithm 4.10 describes the PseudoOP process. Step A, lines 1–2 initialize the local structures to be used. The worst case time regarding the running taken is the same as that of the global data structure initialization which is $O(bs)$. Step A, line 3 calls for a PseudoThrust to do a pseudo-relaxation on a boundary vertex. The PseudoThrust, described in algorithm 4.11, is similar to the MainThrust, but there are two differences. One is that the data structures used in PseduoThrust are both global and local data structures, whereas those in the MainThrust are only global. The other difference is that before pseudo-relaxing super edge $e(x, y, F')$, lines 9–10 confirm that y has not been either closed or pseudo-closed yet, and the current distance or pseudo-distance of y is greater than the pseudo distance obtained from x. Since it takes $O(1)$ time to run lines 9–10, the PseudoThrust CPU time complexity is the same as that of the MainThrust which is $O(b\lg s)$. Lines 4–8 take $O(b)$ time. Consequently, the total CPU time of the iteration step of the PseudoOP process is $O(b\lg s)$, and there is no I/O activity.

Let b_a be the number of PseudoOP processes called for during the execution of the DiskOP algorithm. In other words, b_a is the number of iterations wrt an affected successor fragment, and (b_a+b_u) are the total number of iterations in the DiskOP iteration step. Assume that y is the number of boundary vertices extracted in the *while* loop of the PseudoOP process.

In the iteration step of the PseudoOP process, line 2 extracts a boundary vertex with the minimum distance from a queue consisting of global and local queues, which takes $O(2\lg s+2\lg b)=O(\lg s+\lg b)$. Lines 3–22 determine whether to perform a relaxation or pseudo-relaxation on the extracted vertex, and so on. Obviously, the running time of lines 3–22 is $O(1)$. Lines 23–29 are executed only when the extracted vertex is adjacent to the destination vertex in the query super graph and take $O(1)$ time. Since the CPU complexity of the MainThrust and PseudoThrust is the same, lines 30–38 take $O(b\lg s)$ time. Hence, the total time of the PseudoOP iteration step is $y\times O(b\lg s+\lg s+\lg b)=O(y\times b\lg s)$. Since step B, lines 32 and 34 in the *while* loop lead to reading the DMs y times, I/O bytes is $O(y\times\ B_2)$.

The worst case of step C is that we need to reinsert all the extracted boundary vertices into the global queue. Then, the total running time is $O(y\times b\lg s)$ and there is no I/O activity.

Step D, line 2 is the computation an OPF in a fragment. The CPU OPF's complexity is $O(n\lg n+m)$. Line 1 takes $O(b)$ time, lines 3–11 take $O(b\lg s)$ time and lines 12–14 take $O(b)$ time. Thus, the total running time of PseudoOP, step D is $O(n\lg n+m+b\lg s)$. There is only the one reading fragment by line 2. The I/O bytes is $O(B_1)$.

As a result, the CPU complexity of PseudoOP is $O(y\times b\lg s+n\lg n+m)$. Consequently, that of the b_a PseudoOP process is $O(b_a\times n\lg n+b_a\times m+b_a\times b\times y\lg s)$. The I/O complexity of PseudoOP is $O(y\times B_2+B_1)$, and that of x the PseudoOP process is $O(b_a\times y\times B_2+b_a\times B_1)$.

The total worst case running time complexity of the iteration step in DiskOP is $O(b_u\times b \times\lg s+b_a\times n\lg n+b_a\times m+b_a\times b\times y\lg s)$. The total I/O complexity of DiskOP iteration step is $O(b_u\times B_2+b_a\times y\times B_2+b_a\times B_1)$.

In conclusion, the CPU complexity of DiskOP is $O(b\times s+k_T[p_n\lg p_n+p_m]+s\lg s+k_m+ z\times n\lg n+z\times m+ b_u\times b\times\lg s+b_a\times n\lg n+b_a\times m+b_a\times b\times y\lg s)$, and the I/O complexity is $O(b_u \times B_2+b_a\times y\times B_2+b_a\times B_1+z\times B_1)$.

Chapter 5

Experiments

Chapter 3 and Chapter 4 describe several approaches or heuristics, designed to improve the running time and I/O cost of a disk-based SP algorithm and a disk-based OP algorithm. This chapter focuses on the experiments conducted on a real-world digital map. The testing is conducted on our improved disk-based SP (DiskSPNN) algorithm and the disk-based OP (DiskOP) algorithm. After DiskSPNN is compared with the DiskSP [31], DiskSPN [21] and Dijkstra's algorithm, the performances of DiskOP, NDMA and OPTA are presented.

The PC for testing is a Pentium IV 1.7GHz system with 1GB DDR, and hard drive is an Ultra ATA/100 at 7,200 rpm. The operating system is Microsoft Windows 2000, Server SP4. All the algorithms are developed with Java 1.4.1 with same data structures and are optimized to the same degree. We set the Java Virtual Memory (JVM) at 128M in order to have a uniform environment for all the test cases. Therefore, the main memory is 128M. The digital map that is adopted is the Connecticut road system that is stored in a TIGER/LINE file. We adopt the approach in [31] to convert the data file into a graph consisting of 190,000 edges and 160,000 vertices whose disk file size is approximately 20MB. The partitioning algorithm, proposed in [31], partitioned the graph into fragments. We use a virtual hash table, a disk-based hash table introduced in [31], to hold the fragment database and the DMDB. Some elements in the virtual hash table may not be in the main memory. Let x be the total number of the elements in a virtual hash table and y be the maximum number of the elements in the main memory. In this thesis, we define the cache size of the virtual hash table as $y/x \times 100\%$. When the virtual hash table tries to load an element from the disk and the number of elements in the main memory reaches the maximum, the virtual hash table swaps an element from the main memory to the disk and loads the new element from the disk into the main memory by the LRU strategy. Both the fragment database and the DMDB are memory consuming in the SP and the OP test. The cache size of the databases are critical to the outcome. We set the cache size as low as possible to fit into 64M in order to assume that the proposed algorithms are scalable for the large graphs, which can not be loaded into the main memory as a whole.

The queries for the SP testing are divided into three ranges: long, medium and short, where the long-range queries are more than 66% of the longest possible shortest distance in the graph, the short-range ones are less than 33%, and the medium-range ones are between 33% and 66%. Each range query set for the SP testing has 100 queries. In the OP testing, we are interested in the performance of the three disk-based OP algorithms, not the different ranges of the queries. Therefore, the OP testing query set consists of 30 short-range, 30 medium-range, and 30 long-range queries. All the testing results in this thesis are the average of the queries in the query set.

5.1 Performance of Disk-based SP Algorithms

The experimental results in [21] demonstrate that the fragment of 1000 vertices with the cache size of 22% for the DMDB performs best for the Connecticut graph. It consists of 378 fragments and 347 boundary sets, where 139 fragments are connected to each other, 239 fragments are isolated, and the number of the vertices in isolated fragments is less than 10. In this section, the 1000-node partitioned graph and DMDB with cache size of 22% are used.

In Chapter 3, we presented a new disk-based SP algorithm (DiskSPNN) which is based on the algorithms proposed in [31] and [21]. Our contribution lies in the following, detailed in Chapter 3:

- The DiskSPNN algorithm conducts the skeleton path computation in a query super graph; DiskSP and DiskSPN compute a skeleton path in the merged graph that consists of a source fragment, destination fragment, and super graph;

- DiskSPNN relaxes the outgoing super edges of a closed boundary vertex inside its successor fragment; DiskSP and DiskSPN relax all the outgoing super edges of a closed boundary vertex, indicating that the relaxation involves two or more fragments;
- DiskSPNN prunes the sketch graph with BFS, and DiskSPN prunes the graph by visiting each sketch node in the sketch graph; in addition, DiskSPNN prunes the open boundary vertex during the skeleton path computation;
- DiskSPNN adopts a new data structure, named distance array instead of distance vector used in DiskSP and DiskSPNN; the distance array is almost the same as the distance vector except that the distance array uses an array to hold the boundary vertices in a boundary set instead of a vector.

In this section, we first compare the overall performance of DiskSP, DiskSPN, DiskSPNN, and Dijkstra's algorithm. Then, we discuss the contribution made by each modification in DiskSPNN individually. The programs for testing DiskSP and DiskSPN are the exact same as the ones in [31] and [21], respectively. Dijkstra's algorithm is a main memory version of an SP algorithm. The algorithm loads the digital map into the main memory, and then finds an SP. We assume that the digital map of Connecticut is loaded before executing Dijktra's algorithm. Figure 5.1 demonstrates the running time of different ranges of queries. Disjktra's algorithm is the worst and DiskSPNN is the best of the four algorithms. Figure 5.2 describes the DMDB I/O bytes of the three disk-based SP algorithms. Figure 5.3 presents the average number of boundary vertices that each algorithm accesses during the skeleton path computation phase.

For the short-range queries, the running time of DiskSPNN is only 17% of that of the Dijkstra's algorithm and 29%, and 53% of DiskSP and DiskSPN respectively. For the medium-range, the percentage is 22%, 30%, and 59%, and for the long-range, the time is 29%, 33%, and 60% of Dijkstra's algorithm. Obviously, DiskSPNN significantly reduces the running time for all the queries, especially for the short and medium range queries. The I/O performance is an important aspect of the evaluation of a disk-based algorithm. According to the results of [31] and [21], most of the I/O activities are on a DMDB. The DMDB I/O bytes of DiskSPNN on short and medium queries is only approximately 32% and 63% of the DiskSP and DiskSPN, respectively, the I/O bytes of long queries is 47%

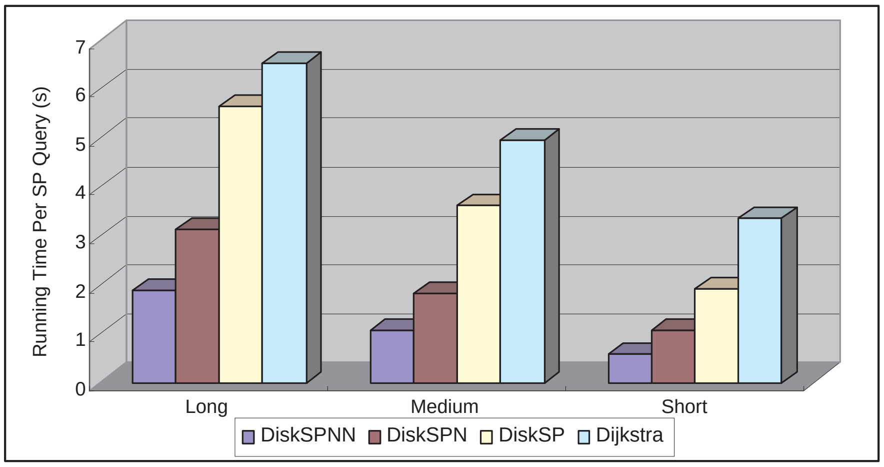


Figure 5.1: Average running time per SP query of four SP algorithms

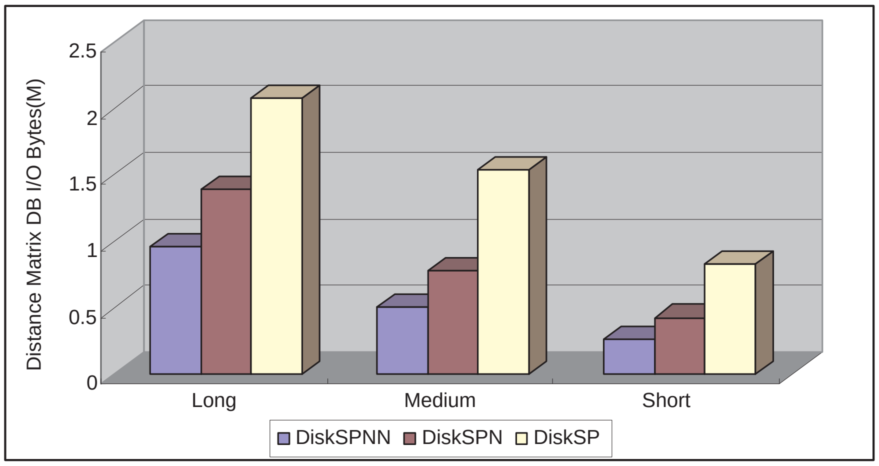


Figure 5.2: Average DMDB I/O bytes per SP query of four SP algorithms

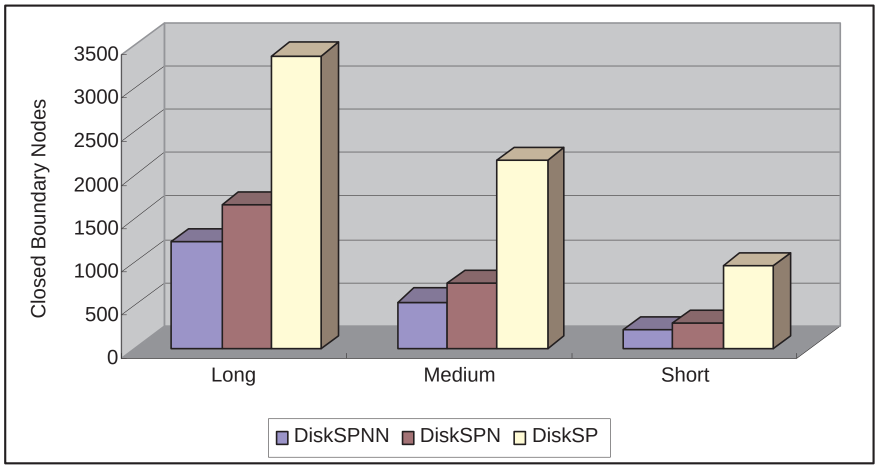


Figure 5.3: Average closed boundary vertices per SP query of four SP algorithms

and 70% of DiskSP and DiskSPN. As a result, the shorter the query, the fewer the I/O bytes, and the less the running time.

During the process to compute a skeleton path, the algorithms close a boundary node, and relax its outgoing super edges during each iteration. Therefore, the number of boundary vertices is the number of iterations of the algorithms. Intuitively, when the size of the search space increases, the number of iterations to compute a query increases. Both DiskSPN and DiskSPNN reduce the search space of the SP queries during the sketch graph pruning phase. In addition, DiskSPNN dynamically prunes the open boundary vertices in the skeleton path calculation phase. The average number of closed boundary vertices of DiskSPN on short, medium, and long queries is 30%, 35%, and 49% of DiskSP respectively. This implies that the search space of the former is less than half of the latter, and the sketch graph pruning is more effective on shorter queries. The pruning of the open boundary vertices further shrinks the search space of DiskSPN. The number of closed boundary vertices of DiskSPNN on short, medium, and long queries is only 24%, 25% and 36% of DiskSP, respectively. The open boundary vertices pruning further reduces 6%, 9%, and 13% search space and works better on longer queries. Consequently, the search space of DiskSPNN is only one third of that of DiskSP. Since the number of loops is independent of the different implementation approaches, the DiskSPNN algorithm is superior to the other two, especially for long queries.

During the sketch graph pruning phase, DiskSPNN prunes with BFS, whereas DiskSPN prunes by visiting each sketch node in DiskSPN (as detailed in Chapter 3). The use of BFS decreases the number of α-approximations. Let n_1 be the number of sketch vertices that DiskSPNN visits, and n be the number of the sketch vertices in the sketch graph. Then, $2n_1$ and $2n$ are the number of α-approximations computed by DiskSPNN and DiskSPN, respectively. In our testing case, the number of α-approximations calculated by DiskSPN is 6.1, 3.2, and 1.7 times that of DiskSPNN for the short, medium, and long-range queries, respectively, as shown in Figure 5.4. Since the α-approximation computations are the most time-consuming part of the sketch graph pruning and the cost of BFS is trivial, the sketch graph pruning of DiskSPN is improved greatly in the DiskSPNN algorithm.

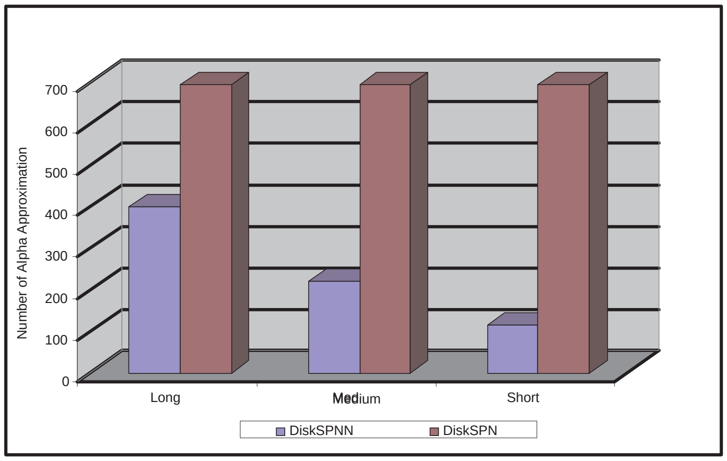


Figure 5.4: Number of α-approximations with PA and PB

DiskSPNN improves four aspects described at the beginning of these section. In order to determine the affect of each aspect, we investigate them separately. Since the last approach changes only a distance vector into a distance array, the improvement is trivial. We investigate it with the query super graph approach. The algorithm based on DiskSP, incorporating the query super graph approach and the distance array data structure are

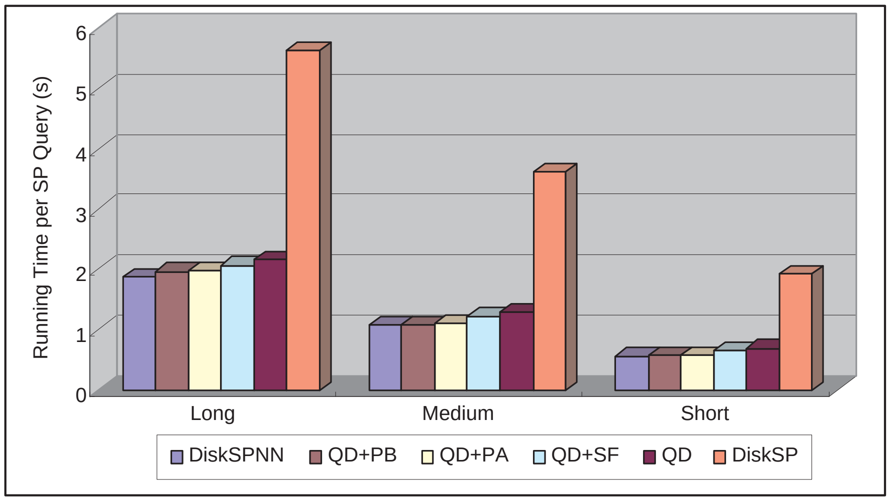


Figure 5.5: Average running time per SP query of different approaches

denoted as *QD*. Continue to apply the successor fragment idea on *QD*, named *QD+SF*. Let *PA* and *PB* be the pruning techniques adopted in DiskSPN and DiskSPNN, respectively. Hence, *QD+PA* and *QD+PB* are the algorithms that apply the different pruning techniques on *QD*. DiskSPNN is the combination of *QD*, *SF*, and *PB*. Figures 5.5, Figures 5.6 and Figures 5.7 give the comparison of these algorithms for the CPU time, and DMDB I/O bytes on the number of closed boundary vertices.

In our testing graph, it is assumed that the number of vertices in a source fragment and destination fragment are 2000, and the number of boundary vertices in the super graph is 3998. Therefore, there are 5998 vertices in the merged graph of DiskSP. However, the query super graph contains only 3998+1+1 =4000 vertices. As known, DiskSP and DiskSPNN compute a skeleton path in the merged graph and the query super graph respectively, and then fill in the skeleton path to obtain the actual SP. In this way, the first approach improves the performance of DiskSP significantly. *QD* improves the running time of DiskSP more than 50% for the all different range queries. *QD+SF* further improve the running time by approximately 4–6% compared with that of *QD*. The DMDB I/O bytes of *QD+SF* are reduced by approximately about 2% of that of *QD*. In Figure 5.6 and Figure 5.7, we find that the pruning techniques of the algorithms decrease the I/O bytes and search space,

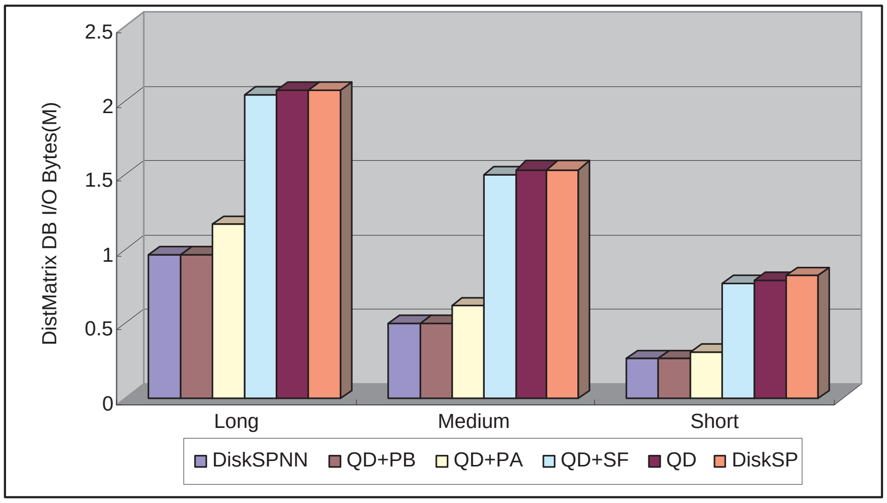


Figure 5.6: Average DMDB I/O bytes per SP query of different approaches

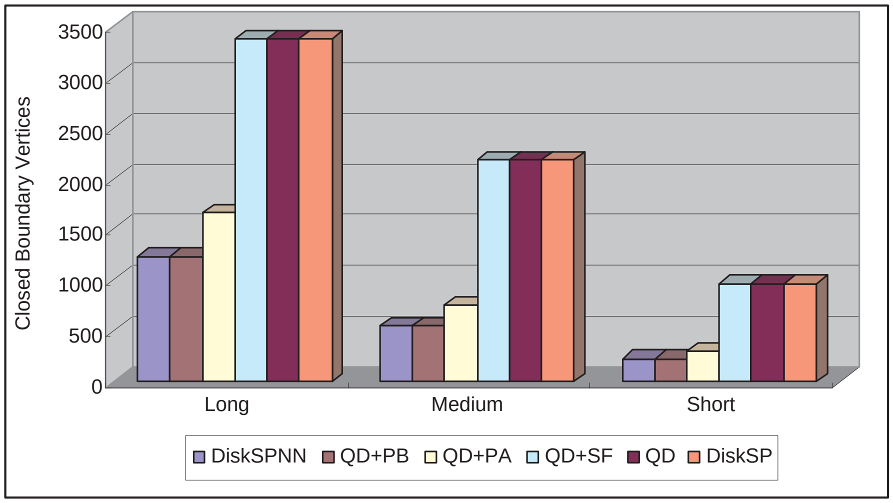


Figure 5.7: Average closed boundary vertices per SP query of different approaches

substantially. As a result, for the short, medium, and long range queries, the running time of *QD+PA* decreases by 15%, 17% and 8% of that of the *QD*, respectively. *QD+PB* improves the running time by 18%, 16%, and 9% for the short, medium-range queries and long-rang queries wrt *QD*. Since the number of accessed boundary vertices of *QD+PB* is at least 20% less than that of *QD+PA*, the search space of *QD+PB* should be 80% of that of *QD+PA*, indicating that the pruning technique of DiskSPNN is better than that of DiskSPN. Since only the pruning techniques can decrease the number of accessed boundary vertices, DiskSP, *QD*, and *SF+ QD* have the same search space. Moreover, the search space of *QD+PA* is the same as that of DiskSPN, and the search space of *QD+PB* is the same as that of DiskSPNN. The DiskSPNN incorporates the successor fragment idea into *QD+PB*, and the running time improves 19%, 17%, and 13% for the short, medium-range queries, and long-rang queries wrt *QD*. According to these experimental results, each approach improves the SP disk-based algorithm individually. Moreover, the approaches work together so that DiskSPNN performs the best of all the algorithms.

Therefore, we can conclude that the overall performance of the DiskSPNN is much better than that of the previous two disk-based algorithms and Dijkstra's algorithm, also, the four approaches proposed in Chapter 3 contribute to the improvement of our disk-based SP algorithm, DiskSPNN.

5.2 Performance of Disk-Based OP Algorithms

Chapter 4 presents three algorithms to solve the OP problem in a very large graph. In this section, we compare the running time, I/O costs of the fragment database and DMDB, and the number of reading each of the two databases for the three disk-based OP algorithms. Furthermore, the best parameters of DiskOP such as the fragment size of the partitioned graph and the cache size of the two databases are investigated. Since the pruning technique in the DiskSPNN can also be applied to the OP problem. Chapter 4 introduces four methods to materialize partial fragments to increase the process to calculate the upper bound of an OP the from a source to a destination. From the performances of them, we choose the best for the sketch graph and open boundary vertex pruning of the OP algorithms. Section 5.2.1 introduces the method for generating the forbidden edge sets in the OP testing, Section 5.2.2 presents the experimental results of the different pivot fragment materialization, Section 5.2.3 finds the optimal parameters to run DiskOP, and Section 5.2.4 compares the three disk-based OP algorithms.

5.2.1 Forbidden Edge Set

For any optimal route query, its OP does not pass through any edge of the forbidden edge set. We generate two types of forbidden edge sets that correspond to the real-world route queries regarding the 1000-node Connecticut graph:

- "Find an OP from A to B not via any tollway". Generally, the express ways are distributed evenly on the road traffic map. As a result, the edges representing tollway consist of the forbidden edge set. There are six forbidden edge testing sets of this type, including 0.01%, 0.1%, 0.5%, 1%, 5%, and 10% of the randomly distributed edges in the graph. Although the 0.01% set covers 17 fragments and the 0.1% covers 103 fragments, the others involve the whole graph that consists of 139 fragments. When more than 10% of the edges in the graph are forbidden, probably there is no OP from a source to a destination;

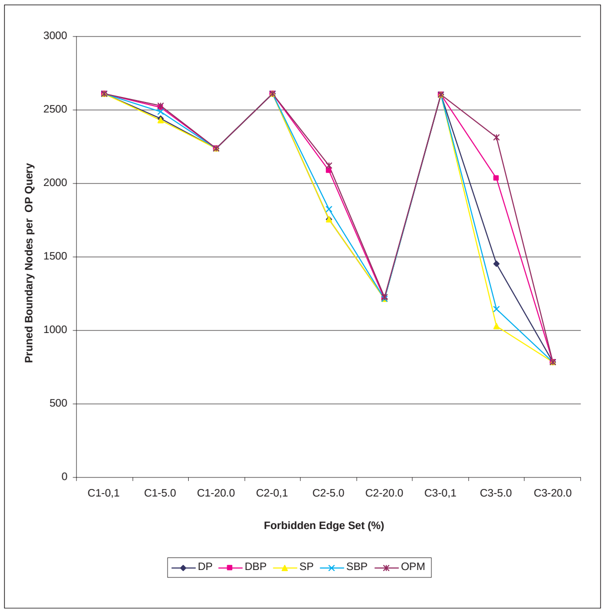
- "Find an OP from A to B not via any traffic jam roads". First, three different affected fragment sets C_1, C_2, and C_3 are generated. C_1 consists of seven adjacent fragments in the middle of the graph. Since, in real-life, traffic jammed roads usually expand to their neighbourhood, $C_A(C, G)$ are defined as a fragment set in partitioned graph G based on another fragment set C. Then, for any fragment F, if F is adjacent to fragment $F' \in C$ and $F \notin C$, then $F \in C_A$. In other words, $C_A(C, G)$ consists of all the fragments that are adjacent to a fragment in set C. Therefore, $C_2 = C_1 \cup C_A(C_1, G)$ and $C_3 = C_2 \cup C_A(C_2, G)$. There are twenty and forty-one fragments in C_2 and C_3, respectively. Let C-x% be a set of edges randomly chosen from each fragment F in C, where x% is the percentage of edges selected from F. In our test cases, x% is 0.1%, 5% or 20%. Finally nine forbidden edge sets C_1-0.1%, C_1-5%, C_1-20%, C_2-0.1%, C_2-5%, C_2-20%, C_3-0.1%, C_3-5%, C_3-20%.

In the following sections, all the experiments are based on these 15 forbidden edge sets. When the partition of the graph changes, for example, the testing graph is a 5000-node Connecticut graph instead of a 1000-node graph, map the preceding edges to the new partition. Therefore the forbidden edge sets are consistent in different testing situations.

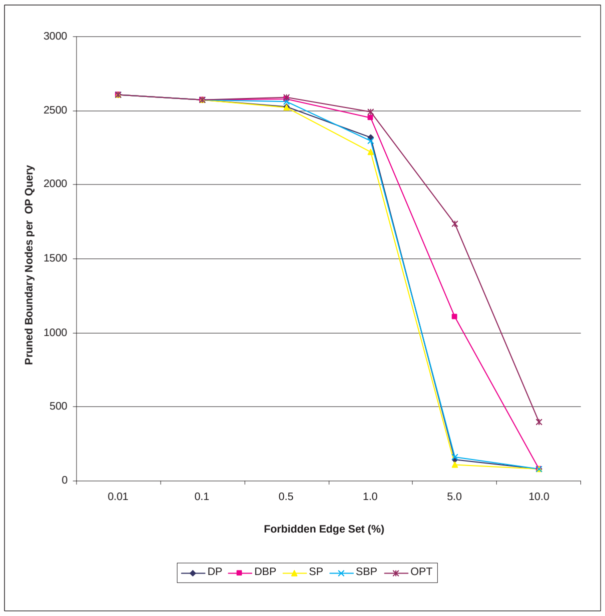
5.2.2 Pivot Fragment Materialization

During the sketch graph pruning phase of an OP computation, some materialized partial fragments, called pivot fragments, are used. The four methods to build a pivot fragment are the *Single Pivot Path*(SPP), *Double Pivot Path* (DP), *Single Branch Pivot Path* (SBP) and *Double Branch Pivot Path* (DBP). In this section, we compare the performance of the four methods in a 1000-node Connecticut graph, and choose one to build a pivot fragment database for testing the disk-based OP algorithms presented in this thesis. The idea is to randomly choose one pivot for each boundary set, and materialize one or two paths for every two pivots in the same fragment. In addition, we use the branch idea to reduce the number of vertices in the pivot fragment (detailed in Chapter 4)

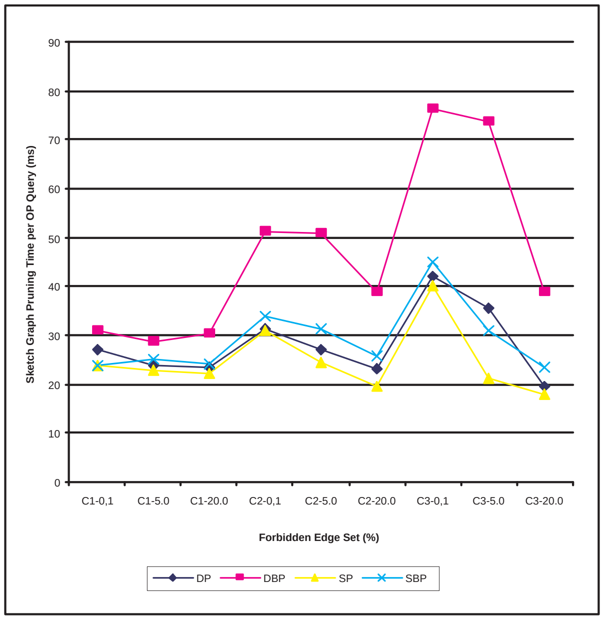
Figure 5.8 illustrates the sketch graph pruning results from the number of pruned boundary vertices and the pruning time aspects. According to Chapter 4, we use pivot fragments to calculate the γ-approximation of an OP query. since γ-approximation is an upper bound of the optimal path's weight, the tighter the bound is, the more the boundary vertices in the sketch graph are pruned. When all of the fragments are affected, we need to re-compute the weights of all the sketch edges with pivot fragments (detailed in Chapter 4). In Figure 5.8 (a)–(b), there is another approach called the OPM, to calculate the weights of the affected sketch edges by reading the entire affected fragment into the main memory. Obviously, the γ-approximation of the OPM is the best case for the pivot fragment materialization; however the I/O cost is too high. So, in practice, OPM is never used. In Figure 5.8(a)–(b), the number of pruned boundary vertices decreases as the percentage of forbidden edges (*PFE*) increases, since the computed γ-approximation, based on the materialized pivot fragments, is no longer close to the optimal distance. When the forbidden edges are distribute in the entire graph and the *PFE* is greater than 2%, the pruning technique may not be effective. If the *PFE* closes to 10%, it is almost



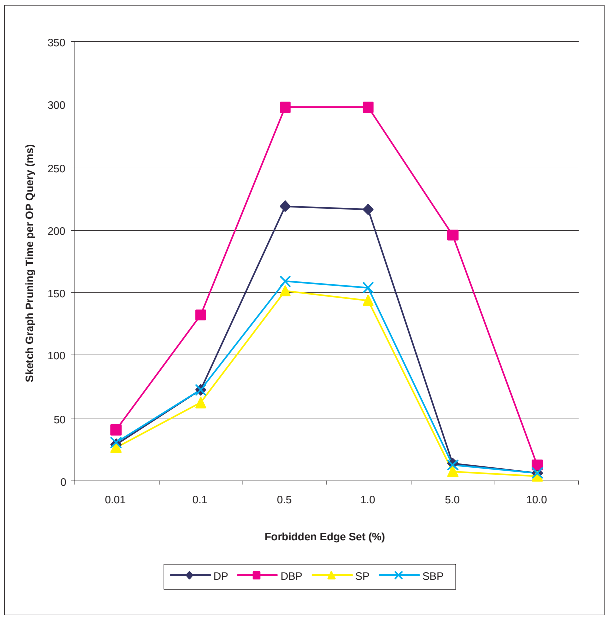
(a)



(b)



(c)



(d)

Figure 5.8: Performance of partial fragment materializations

Materialization Name	***DP***	***DBP***	***SP***	***SBP***	***Fragment Database***
File Size	1.33M	1.34M	0.68M	0.74M	19.5M

Table 5.1: File size of pivot fragment materialization

impossible to prune the sketch graph. When the forbidden edges are concentrated on some area of the graph, the pruning technique is effective. In Figure 5.8(a), more than 700 boundary vertices are pruned, even for the forbidden edge set C_3-20%. Since there are 3,998 boundary vertices in the 1000-node Connecticut graph, 20% of the area is pruned. Consequently, that the pruning technique is useless when the number of forbidden edges is large, and are distributed evenly on the whole graph.

All the partial fragment approaches work well when the *PFE* is small, and the difference in the pruned boundary sets among them are trivial. Since the *PFE* in the whole graph is greater than 1% or the *PFE* inside a cluster of fragments is approximately 5%, the DBP is the one close to the OPM, and obviously is better than the others. For example, the DBP prunes more than 1000 boundary vertices than the other three approaches for the forbidden edge set 5.0%.

Figure 5.8(c)–(d) indicate that the running time to prune a sketch graph. The time of the SBP and SPP is about 30% less than that of DBP and DP. Therefore, the SBP is recommended only when few edges are forbidden in the graph (e.g., 0.5%). Otherwise, the DBP is recommended. Table 5.1 depicts the file sizes of the pivot fragment databases that are materialized by the previous approaches. Among them, the size of the SPP is the minimum, and that of the DP is the maximum, the DBP is only 6.5% of the size of the fragment database. The size of the DBP is slightly less than that of DP.

Although the running time and file size of the DBP are worse than those of SBP and SPP, respectively. DBP prunes many more boundary vertices than others do. For the optimal route query problem, the more boundary vertices pruned, the fewer affected fragments involved, and the less the DM re-computation is needed. By using the DBP, we spend 0.1s more than with the other approaches during the sketch graph pruning, and save at least ten times the running and I/O time during the skeleton path calculation. In the next section, the experimental result indicates that the time to prune a sketch graph is less than 10% of the skeleton path computation. Hence, by considering the cost and benefit, DBP is superior, when the number of forbidden edges is large. If only a few edges are forbidden, the *SBP* is the best. In order to simply the test cases, we use the DBP pivot fragment materialization in our disk-based OP algorithm.

5.2.3 Optimal Parameters of the DiskOP Algorithm

In the DiskOP algorithm , since both the fragment database and the DMDB are stored in virtual hash tables, their cache sizes are critical to the outcomes. For a forbidden edge set and a graph, the smaller the fragment size is, the lower the percentage of affected fragments. However, the increase in the number of fragments can cause more CPU computations due to the increased number of PseudoOP processes. Therefore, the fragment size of the partitioned graph is also very important to the performance of DiskOP. First we find the optimal fragment size of the Connecticut graph. Then, we figure out the best cache size for the fragment database and the DMDB, respectively. Since the three parameters are variable, our approach is to choose two parameters to obtain the optimal value of the other parameter.

Optimal Fragment Size

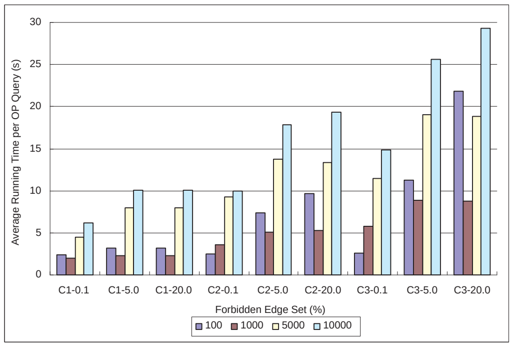
Since the main memory is large enough to load all the fragments and their DMs of the Connecticut Graph simultaneously, we set the cache sizes of the virtual hash tables holding the two databases, at 100%. Therefore, it is easy to find out the optimal fragment size by ignoring the performance of I/O. After comparing the performance of DiskOP in the partitioned graphs with the fragment sizes of 100-node, 1000-node, 5000-node, and 10000-node (Figure 5.9), we choose the best one as the optimal fragment size of the Connecticut graph. The numbers of fragments in the above graphs are in Table 5.2. Since there are some isolated vertices in the Connecticut graph, the partition algorithm in [31] groups them in the isolated fragments which have no boundary vertices. In this thesis, we consider only those connected fragments, and the number of fragments in a graph refers to the number of connected fragments.

Fragment size of CT Graph	# of Fragments	# of Connected Fragments
100 -node	1693	1452
1000 -node	378	139
5000 -node	268	29
10000 -node	254	15

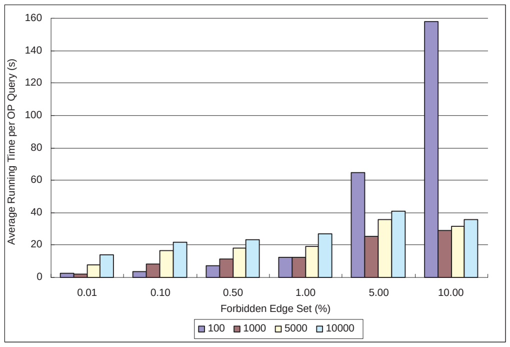
Table 5.2: Number of fragments in the Connecticut graphs with different fragment size

Figure 5.9(a)–(b) reveal the average OP query running time in the 100-node, 1000-node, 5000-node, and 10000-node Connecticut graphs. For the forbidden edge sets C_2-0.1%, C_3-0.1%, 0.01%, 0.1%, and 0.5%, the 100-node graph is the best, whereas the 1000-node graph is the best for all the others. In Chapter 4, we know that a boundary node is either closed in DiskOP **while** loop or closed/pseudo-closed in the PseudoOP process called by DiskOP. Thus, its adjacent super edges inside its successor fragment is relaxed/pseudo-relaxed. As a result, the total times of closing/pseudoclosing the boundary vertices are the number of accessed boundary vertices in DiskOP. Figure 5.9(c)–(d) relate that the number of accessed boundary vertices in the 100-node graph is much greater than that in the other graphs except for the forbidden edge sets C_1-0.1%, C_2-0.1%, C_3-0.1%, 0.01%, 0.1%, and 0.5%.

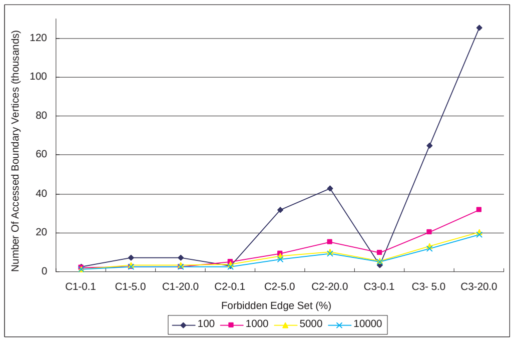
According to Table 5.2, the number of fragments in the 100-node graph is far more than the number in the other graphs. As a result, there are more boundary vertices in the the 100-node graph than in the others. However, the percentage of affected fragments (PAF) the in 100-node is much lower than those of the other graphs (figure 5.9(e)–(f)) for the forbidden edge sets C_2-0.1%, C_3-0.1%, 0.1% and 0.5%.When the PAF in the graph is low, the number of PseduoOP processes should be reduced. As a result, the number of accessed boundary vertices in the 100-node graph is close to that of the other graphs for C_2-0.1%, C_3-0.1%, 0.1% and 0.5% forbidden edge sets as shown in Figure 5.9(c)–(d).



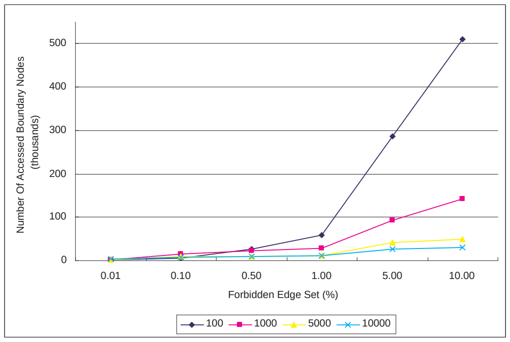
(a)



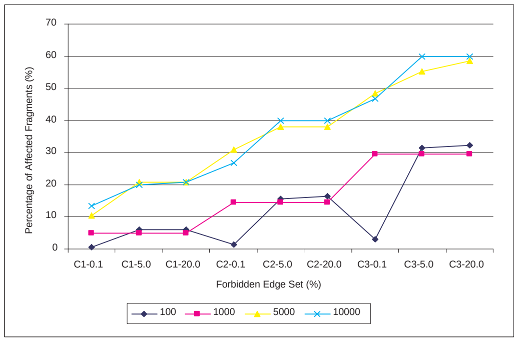
(b)



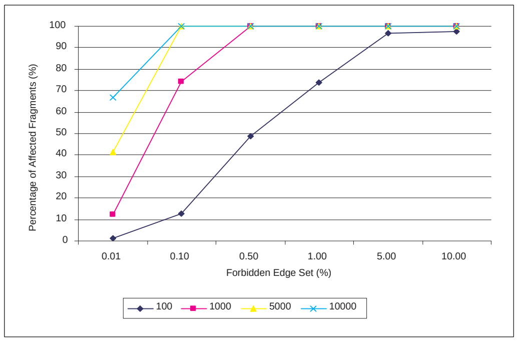
(c)



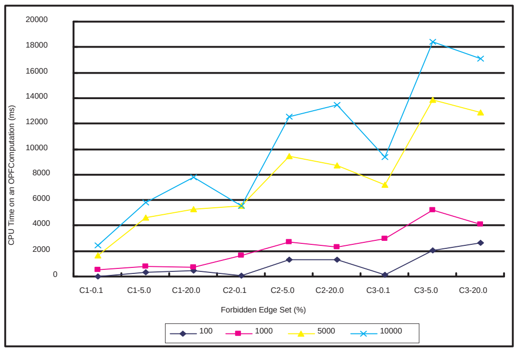
(d)



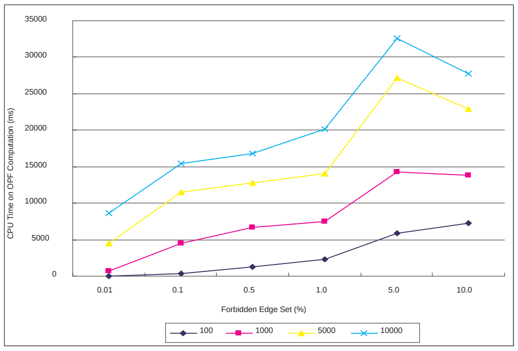
(e)



(f)



(g)



(h)

Figure 5.9: Performance of DiskOP in partitioned graphs with different fragment sizes

The most time consuming tasks of DiskOP are the relaxations on the boundary vertices and the OPF computations during the PseudoOP processes. The first task involves accessing of boundary vertices previously discussed. Figure 5.9(g)–(h) exhibit the total running time of OPF computations per OP query inside the 100-node, 1000-node, 5000-node, and 10000-node graphs. In spite of the fewer boundary vertices in a 5000-node graph and 10000-node graph than in a 1000-node graph, the DiskOP Algorithm spends much more time on the OPFs computation in the two graphs than in the 1000-node graph. Consequently, DiskOP performs much worse in the 5000-node graph and 10000-node graph than in the 1000-node graph. The OPF time in the 100-node graph is less than that in the 1000-node graph for all the forbidden edge sets. For the forbidden edge set C_2-0.1%, C_3-0.1%, 0.1% and 0.5%, the time on the relaxations of boundary vertices in both graphs are almost the same. Therefore, the running time of an OP query in the 100-node graph is less than that in the 1000-node graph. For the other forbidden edge sets, the relaxation time on the boundary vertices in the 100-graph is much more than that in the 1000-graph. In addition, the difference in the OPF time in the two graphs is not very large. Consequently, the OP query runs faster in the 1000-node graph than in the 100-node graph.

From these experimental results and analysis, we can conclude that for C_2-0.1%, C_3-0.1%, 0.1%, and 0.5% forbidden edge sets, DiskOP works best in the 100-node graph and 1000-node graph. For C_1-0.1%, C_1-5.0%, C_1-20.0%, C_2-5.0%, C_2-20.0%, C_3-5.0%, C_2-20.0%, 0.01%, 1.0%, 5.0% and 10.0% forbidden edges sets, the 1000-node graph is the best choice. Since the performance on the 1000-node graph is good for all the forbidden edges, in order to simplify the testing in the following sections, we choose the fragment size, 1000, as the optimal value. The remaining experiments are tested in the 1000-node Connecticut graph.

Optimal Cache Size of the Fragment Database

To obtain the optimal cache size of the fragment database, we maximize the cache size of the DMDB and execute the DiskOP algorithm in the 1000-node Connecticut graph with fragment database cache sizes: 7%, 14%, 22%, 36%, 50%, and 72%.

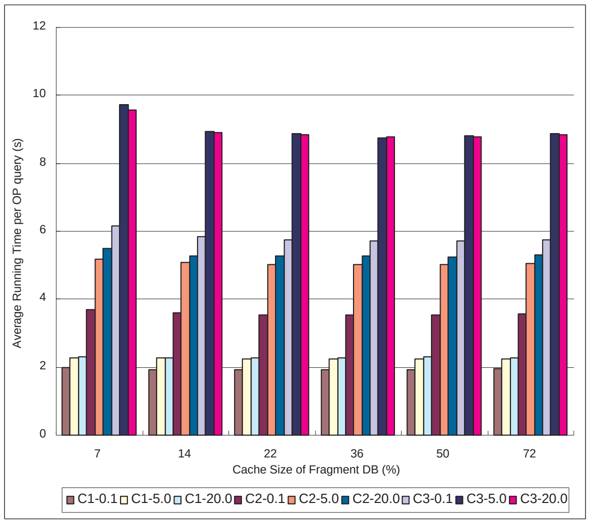
Figure 5.10 displays the performance of DiskOP with different cache sizes of the fragment database regarding the running time aspect and fragment database I/O bytes. Obviously, the performance of DiskOP improves with the increase of cache size. According to Figure 5.10(c)–(d), after the cache size reaches 22%, the fragment database I/O bytes are almost unchanged. Moreover, there is no obvious decrease tendency in the running time of DiskOP, when the cache size is greater than 22%. Therefore, 22% is the optimal cache size of the Connecticut 1000-node fragment database.

We find that DiskOP works well even when cache size is as small as 10%. The difference in the running time and fragment database I/O bytes is trivial, if the forbidden edges in the graph are sparse. Even when the percentage of forbidden edges is greater than 5%, the maximum fragment I/O bytes are fewer than 30M. Thus, we can infer that algorithm DiskOP is scalable in a very large graph.

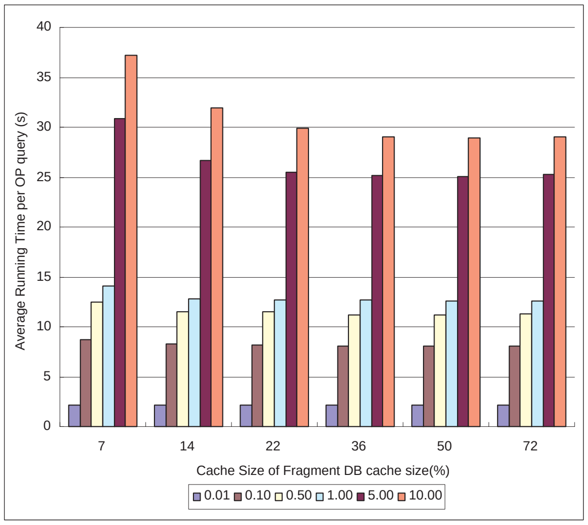
Optimal Cache Size of DMDB

Similarly, the optimal cache size of the DMDB is found by maximizing the cache size of the fragment database and executing the DiskOP algorithm in the 1000-node Connecticut graph with the DMDB cache sizes 7%, 14%, 22%, 36%, 50%, and 72%. Figure 5.11 exhibits the performance of DiskOP with different DM cache sizes from the running time and I/O aspects.

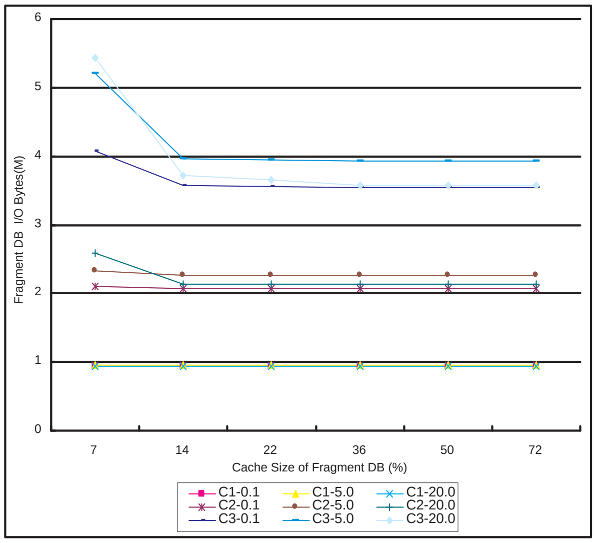
The cache size of DMDB is critical to the performance of DiskOP according to the experimental results in Figure 5.12. The difference of the I/O DMDB bytes is approximately 700M for cache sizes 7% and 50%. When the cache size reaches 50%, the improvement is trivial. Fortunately, the size of DMDB is far smaller than that of fragment database. For example, the cache size of DMDB is only 11% of that of fragment database for the 1000-node Connecticut graph. Therefore, the optimal cache size of the DMDB in our test is selected as 50% does not cause a main memory problem.



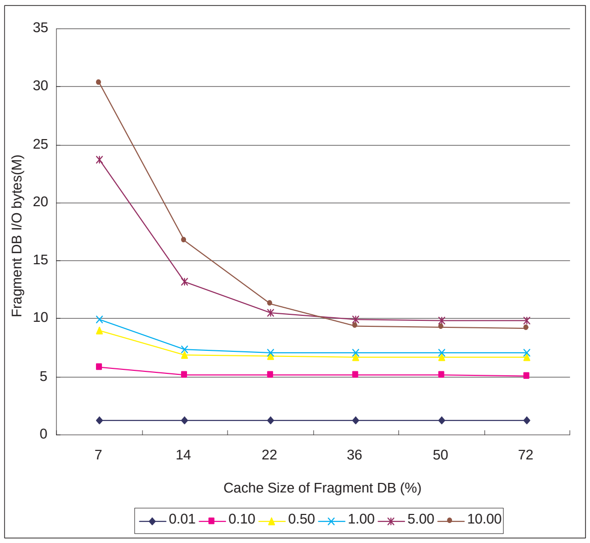
(a)



(b)

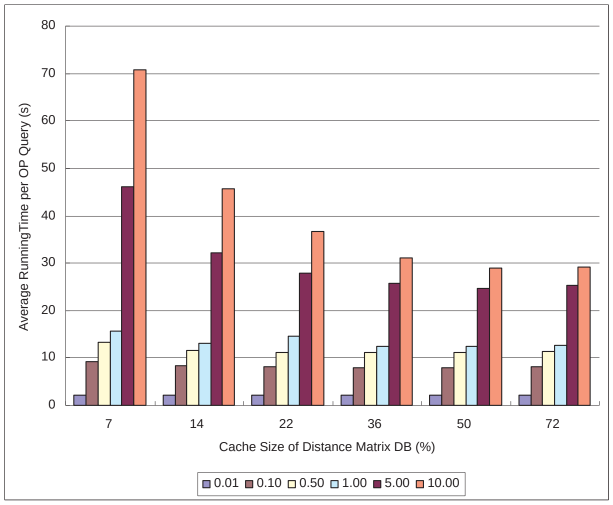


(c)

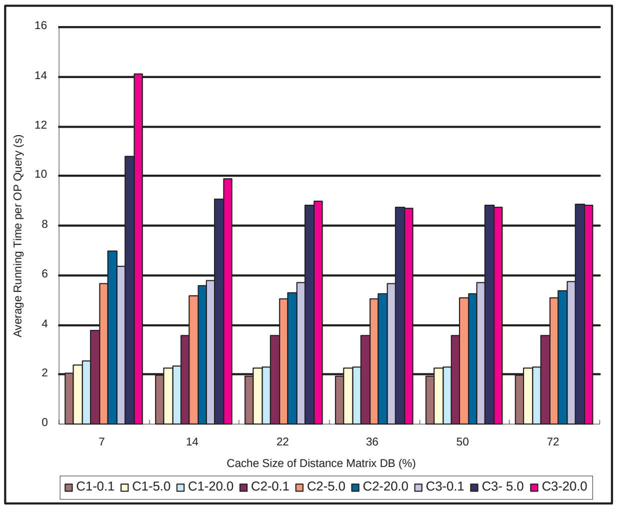


(d)

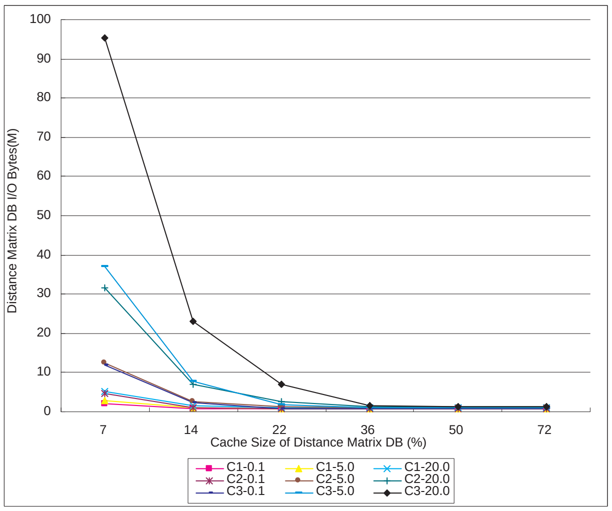
Figure 5.10: Performance of DiskOP with different fragment database cache sizes



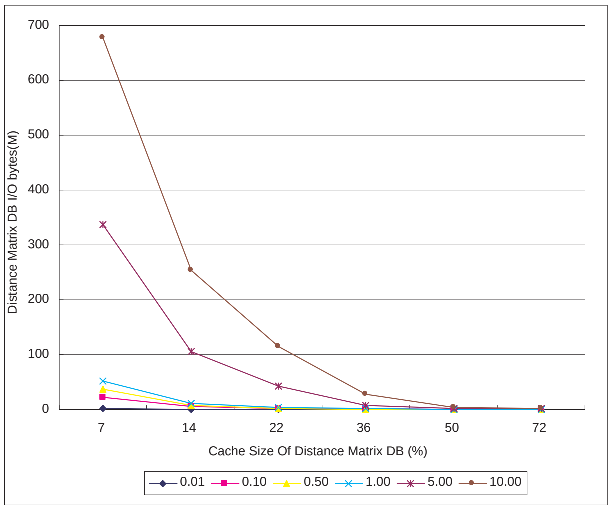
(a)



(b)



(c)



(d)

Figure 5.11: Performance of DiskOP with different DMDB cache sizes

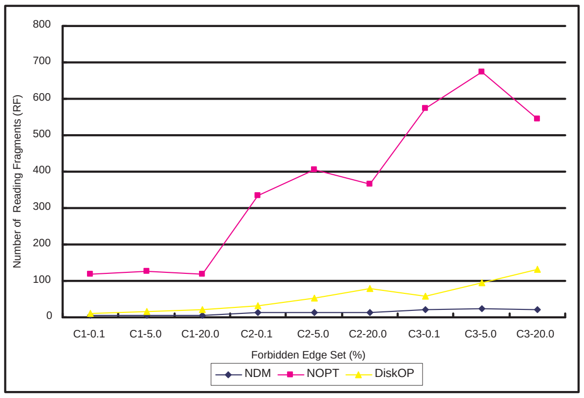
5.2.4 Disk-Based OP Algorithms

In the previous sections, the optimal parameters are obtained for the obtained fragment size, the cache size of the fragment database, and the cache size of the DMDB. In this section, we focus on the comparison of the three algorithms proposed in Chapter 4. The experimental results of DiskOP are with the three optimal parameters: fragment size 1000 nodes, fragment database cache size 22%, and DMDB cache size 50%. In order to show that DiskOP is superior to the other two algorithms, the experimental results of both NDM and NOPT are computed with fragment database cache size 100%, and DMDB cache size 100% in the 1000-node graph. Therefore, both of the databases in NDM and NOPT are loaded into the main memory.

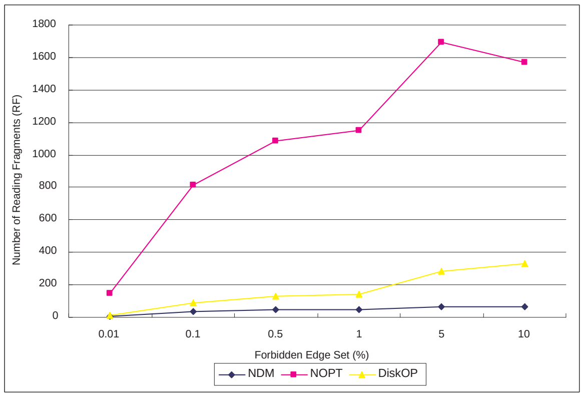
The comparison consists of five aspects: the average running time of optimal queries, the average fragment database I/O bytes, the average DMDB I/O bytes, the average number of the reading fragments RF(A), and the average number of the reading DMs RD(A) in the skeleton path computation phase, where A is the name of the algorithm executed.

Let n_l be the number of loops in the algorithms, n_{ua} be the number of the extracted boundary vertices with an unaffected successor fragment and n_a be the number of the extracted boundary vertices with affected successor fragment in the **while** loop. Then RF(NOPT)=n_a, RD(NOPT)=n_{ua} and RF(NDM)=f_a, RD(NDM)=n_l , where n_l=n_a+n_{ua} and f_a is the number of affected fragments in the **while** loop. Obviously, RF(NDM)<< RF(NOPT). Since NDM re-computes the DMs for those affected fragments and puts it in temporary database, RD(NDM) is the sum of n_a and n_{ua}. RF(DiskOP)$\leq$ n_a because DiskOP reads a fragment only when the extracted boundary vertex has an affected successor fragment and there exist open boundary vertices in the fragment. RD(DiskOP)=n_{ua} +n_{pseudo}, where n_{pseudo} is the sum of the times reading DMDB during the PseudoOP processes, when the boundary vertices with the affected successor fragments are extracted. Therefore RD(DiskOP)>RD(NDM)>RF(NOPT). In the following paragraphs, we verify the analysis by our experimental results.

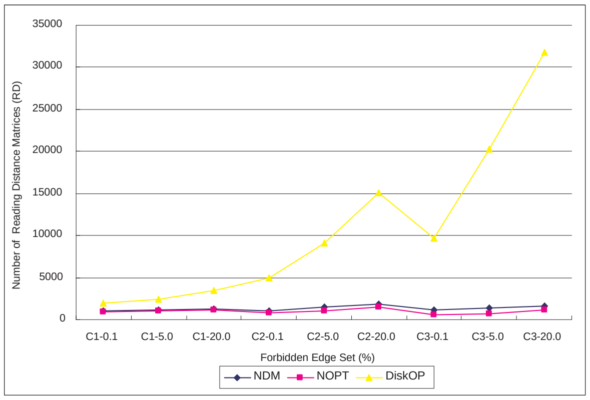
Figure 5.12(a)–(b) depict the average number of reading fragments in DiskOP, NDM, and NOPT algorithm. Whatever the type of a forbidden edge set, the number of reading fragments in NOPT is much greater than that in the other two algorithms. Although, for all forbidden edge sets, RF(NDM)<RF(DiskOP), the differences small, indicating that the



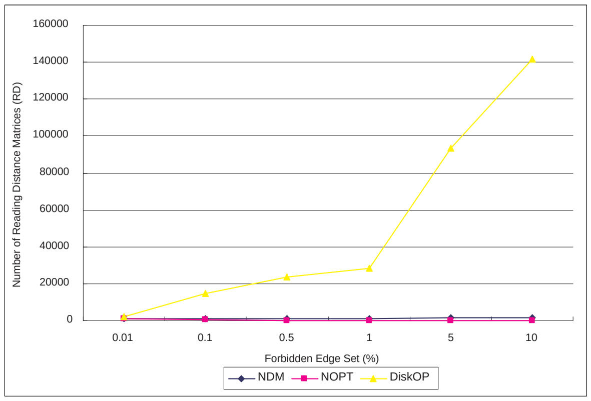
(a)



(b)



(c)



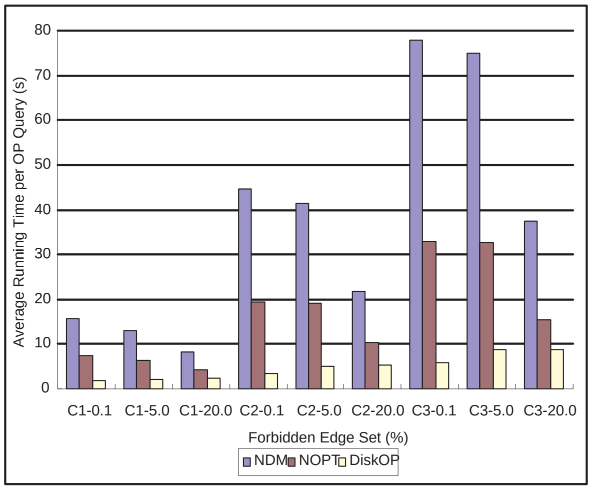
(d)

Figure 5.12: Number of reading fragments and DMs for DiskOP, NDM, and NOPT

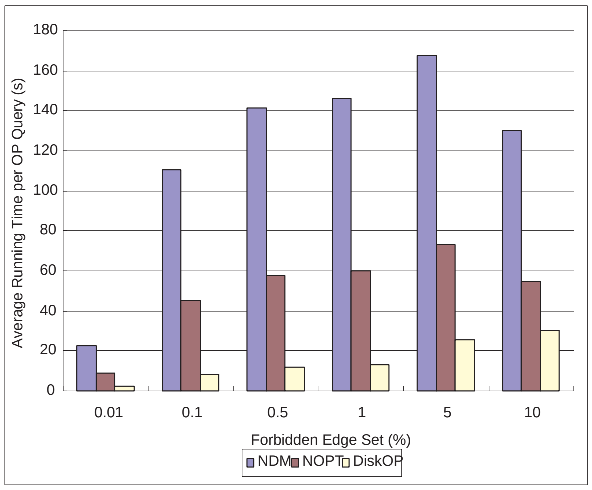
DiskOP algorithm performances well on fragment database I/O. Figure 5.12(c)–(d) gives the average number of reading distance matrices in the three algorithms. The experimental data indicates RD(DiskOP)>>RD(NDM)>RD(NOPT). Fortunately, the size of DMDB is only around 10% of that of fragment database. According to the test in Section 5.2.3, when the DMDB cache size reaches 50%, the DiskOP works as well as loading the entire database into the main memory.

As the size of the graph increases, the use of the optimal fragment database cache size can cause trouble. For example, for the fragment database with size 200M, reading 50% of the fragments into the main memory, simultaneously, consumes apprixIMATELY 200×50%×3 = 300M by assuming that the main memory for loading database is three times its disk file size. Since many PCs cannot handle this, the only solution is to reduce the cache size of fragment database. Because the size of DMDB is approximately 10% of the size of fragment database, reading only 50% of the distance matrices into the main memory at the same time consumes about 200×10%×50%×3 = 30M. Therefore, we can use the optimal value of the DMDB cache size, but must adjust that of the fragment database cache size. As a result, although the number of reading DMDB in DiskOP is large, DiskOP still performs well in DMDB I/O. Figure 5.13(e)–(f) proves it. The worst DMDB I/O in our test cases of DiskOP is less than 5M. Figure 5.13(c)–(d) portray that fragment database I/O bytes of DiskOP are close to those of NDM and NOPT for all the forbidden edge sets, where only 22% of the fragments in DiskOP are in the main memory, and all the fragments in NDM and NOPT are in the main memory. Consequently, the theoretical analysis and experiment results confirm that the I/O performance DiskOP is superior.

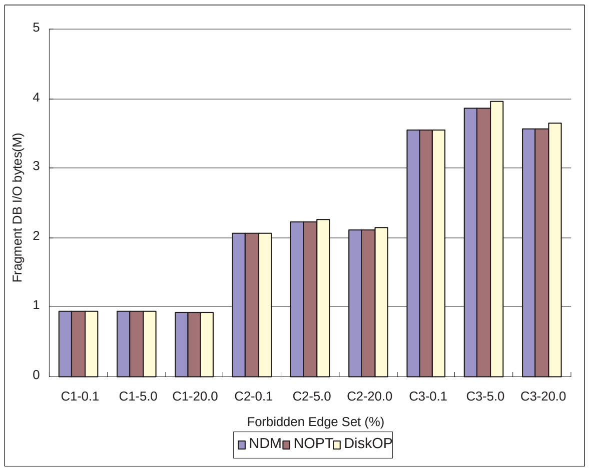
According to the algorithms' description in Chapter 4, the RF(DiskOP) is the number of OPFs computed in DiskOP, the RF(NOPT) is the number of OPTs calculated in NOPT, and the RF(NDM) is the number of DMs re-computed in NDM. Let t be the time to build an OPT. Since the time on an OPT computation is almost the same as that on the OPF, the total time of the OPF computation in DiskOP is RF(DiskOP)×t, the total time of the OPT computation in RF(NOPT)×t. Let b be the average number of boundary vertices inside a fragment. Then, the total time of DM re-computation of NDM is b×RF(NDM)×t. As a result, RF(DiskOP)×t<<b×RF(NDM)×t and RF(DiskOP)×t<<RF(NOPT)×t. Based on the previous analysis on CPU cost as well as I/O cost, we can conclude the average



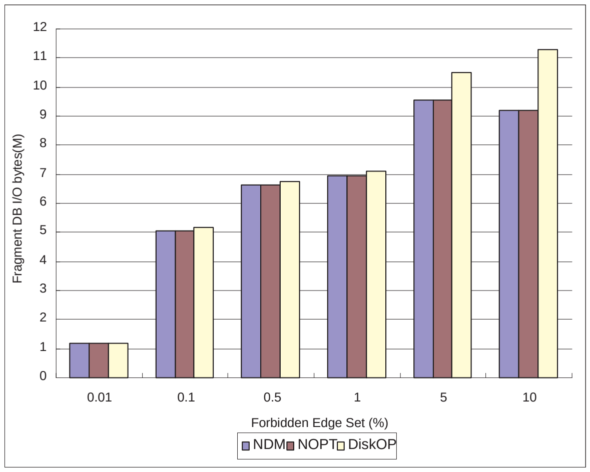
(a)



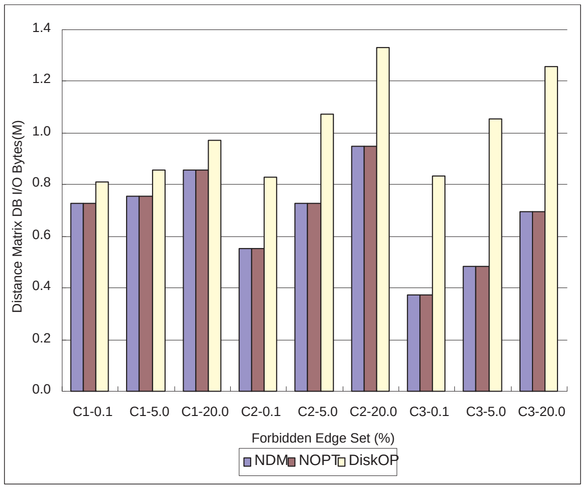
(b)



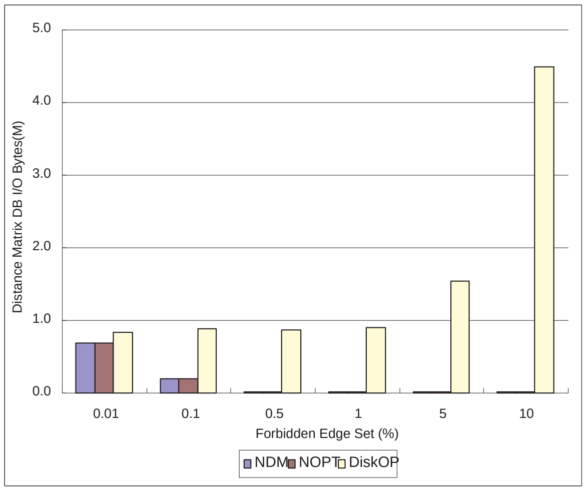
(c)



(d)



(e)



(f)

Figure 5.13: Performances of DiskOP, NDM, and NOPT

running time of DiskOP is much less than that of the other two algorithms. Figure 5.13(a)–(b) provides the experimental results to prove it.

From the analysis and the experimental results in this section, it is evident that the proposed DiskOP algorithm is superior to the both NDM and NOPT.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In this thesis, several approaches to speed up the previous disk-based shortest path algorithms, and a new disk-based optimal path algorithm have been presented. The new disk-based SP algorithm (DiskSPNN) and our disk-based optimal path algorithm (DiskOP) are based on the framework of the DiskSP algorithm and DiskSPN algorithm ([31],[21]) which are derived from Dijskstra's algorithm. The essence of the disk-based algorithms is to divide a large graph into a set of fragments, and pre-compute the distance matrices containing the shortest distances between each pair of boundary vertices inside each fragment. In order to prune the searching space, a boundary set distance matrix is also pre-computed to hold the maximum and the minimum shortest distances between each pair of boundary sets. When a route query is calculated, these pre-computed data are adopted to speed up the whole process. There are several steps in a route query calculation process: sketch graph pruning, finding a skeleton path, and filling out the skeleton path.

For the shortest path problem, we reduce the number of α-approximations significantly with the breadth first search in the sketch graph pruning phase; then we compute a skeleton path in a query super graph instead of a merged super graph. This avoids the unnecessary relaxation of the super edges inside the successor fragment of a closed boundary vertex, and dynamically prunes some open boundary vertices during the finding skeleton path phase. The experimental results demonstrate that our SP algorithm, DiskSPNN, reduces the calculation time as well as the I/O bytes.

For the optimal route query problem, due to the forbidden edges, a β-approximation is invalid for estimating the upper bound of the optimal distance. As a result, we materialize a pivot fragment database to calculate the γ-approximation used in pruning the sketch graph and open boundary vertices. With minor modifications, the techniques in DiskSP, including the query super graph, successor fragment, and pruning the sketch graph and open boundary vertices, are adapted to the optimal path problem. In addition, we present a new approach, pseudo-relaxation and co-relaxation to balance the calculation time and I/O activities. The sketch graph pruning phase of DiskOP is the same as that of DiskSPNN except for the γ-approximation instead of the β-approximation. Both DiskSPNN and DiskOP fill out the skeleton path by computing an actual shortest/optimal path for each super edge of the skeleton path.

In the skeleton path computation phase, when closing a boundary vertex inside an unaffected successor fragment, DiskOP works like DiskSPNN; however, if successor fragment F of closed boundary vertex b contains forbidden edges, DiskOP first continues the optimal skeleton path tree computation by ignoring the forbidden edge constraint until all the boundary vertices in F are included in the tree. Then, the algorithm simultaneously relaxes the closed but un-relaxed boundary vertices inside F by computing an optimal path forest. Consequently, DiskOP finishes the relaxation of a set of boundary vertices with a partial skeleton path tree computation, building one optimal path forest, and one fragment reading. Although the computing and maintaining of the partial skeleton path trees leads to the reading of distance matrices and some calculations, the benefits make it worthwhile to do. In this way, DiskOP balances the calculation time and I/O activities. Since little research has been done in the area of disk-based optimal path algorithms, we compare the DiskOP algorithm with two brute-forces approaches. One approach re-computes the distance matrix of the affected fragments in real-time, and the other relaxes closed boundary vertex b with affect fragment F by building an optimal path tree rooted b inside F.

The calculation time of the first approach is untolerable, but it minimizes the I/O activities. Although the calculation time of the second approach is acceptable, its I/O performance is unacceptable, when the graph is very large and most of the fragments contain forbidden edges. The experimental results prove that DiskOP performs well for the calculation time and I/O activities under various forbidden edge distributions. Even when DiskOP is run with restricted main memory resources and two brute-force approaches are run in main memory, DiskOP is still superior. This indicates that the calculation time of DiskOP is much less than that of the two approaches. Due to frequent skeleton path trees computation in DiskOP, the I/O activities on the distance matrix database of DiskOP are worse than that of the two approaches. Fortunately, the size of the distance matrix database is far less than that of the fragment database. We can solve the problem by increasing the cache size of the virtual hash table holding distance matrix database. The experimental results also confirm that DiskOP works well with minimum memory (e.g., cache size 10% and 50% of fragment database and distance matrix database, respectively, in the 1000-node Connecticut graph). Therefore, we can infer the scalability of our DiskOP algorithm is good.

In summary, we improve the existing disk-based shortest path algorithms by various techniques without new materialization. The DiskSPNN algorithm requires even less main memory than the previous disk-based shortest path algorithms; the DiskOP algorithm works efficiently, and is applicable to very large graphs.

6.2 Future Work

In this thesis, we have presented the algorithms to answer the shortest/optimal path queries in very large spatial databases quite expediently. Our research on the optimal route query problem focuses only on the forbidden edge constraint. However, in the real world, the query constraints are more complex than just those about roads or areas.

By studying various query constraints, we find the forbidden edge constraint is the most basic one. Many constraints can be reduced to the forbidden edge constraint such as finding a simple optimal path from city A to B via C and D in order. Intuitively, an optimal path consists of three optimal sub-paths. The first one is an optimal path from A to C not via city B and D, the second sub-path is an optimal path from C to D, not via city A, B and any edge on the first path; and the last sub-path is an optimal path from D to B not via city A, C, and any edge on the first two sub-paths. Therefore, the previous constraints become the forbidden edge constraint. In the future, we will develop a framework to answer optimal paths queries with various constraints in a very large graph.

Another practical problem is to process multiple optimal path queries in near-real time. Some techniques are proposed in [21] for the multiple SP, but further work is required for the optimal path queries (e.g., adapting the techniques in [21] to the optimal path problem).

Bibliography

[1] Y. P. Aneja, V. Aggarwal, and LPK Nair. *Shortest chain subject to side constraints.* Networks, 13(2):295–302, 1983.

[2] L. Arge. *The buffer tree: a new technique for optimal I/O-algorithms.* In Proc. Workshop on Algorithms and Data Structures, LNCS 955:334-345. Springer-Verlag, Berlin, 1995.

[3] L. Arge, G. Brodal, and L. Toma. *On external-memory MST, SSSP and multiway planar graph separation.* Journal of Algorithms, November 2002.

[4] Surender Baswana, Ramesh Hariharan, and Sandeep Sen. *Improved decremental algorithms for maintaining transitive closure and all-pairs shortest paths.* STOC:117–123, 2002.

[5] J. E. Beasley and N. Christofides. *An algorithm for the resource constrained shortest path problem.* Networks, 19:379–394, 1989.

[6] Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, and Clifford Stein. *Introduction to algorithms.* McGraw-Hill Book Company, 2001.

[7] Camil Demetrescu and Giuseppe F. Italiano. *Fully dynamic all pairs shortest paths with real edge weights.* FOCS:260–267, 2001.

[8] Camil Demetrescu and Giuseppe F. Italiano. *A new approach to dynamic all pairs shortest paths.* STOC:159–166, 2003.

[9] M. Desrochers and F. Soumis. *A generalized permanent labeling algorithm for the shortest path problem with time windows.* INFOR, 26:191–212, 1988.

[10] Irina Dumitrescu. *Constrained path and cycle problem.* a Ph.D. thesis presented to the University of Melbourne in Mathematics and Statistics, 2002.

[11] D. Eppstein. *Finding the K shortest paths.* SIAM J. Computing 28(2):652–673, 1999.

[12] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. *Semi dynamic algorithms for maintaining single source shortest path trees.* Algorithmica 22(3):250–274,, 1998.

[13] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. *Fully dynamic algorithms for maintaining shortest paths trees.* Journal of Algorithms, 34:351–381, 2000.

[14] M. R. Garey and D. S. Johnson. *Computers and intractability: a guide to the theory of NP-completeness.* Freeman, San Francisco, 1979.

[15] G. Y Handler and I. Zang. *A dual algorithm for the constrained shortest path problem.* Networks 10:293–306, 1980.

[16] John Hershberger, Subhash Suri, and Amit Bhosle. *Finding the K shortest simple paths.* Technical Report, University of California, Santa Barbara, 2001.

[17] B. Jaumard, F.Semet, and T. Vovor. *A two-phase resource constrained shortest path algorithm for acylic graphs.* Les Cahier du GERAD, G96-48, 1996.

[18] Victor M. Jimenez and Andres Marzal. *Computing the K shortest paths: a new algorithm and an experimental comparison.* Proc. 3rd Workshop. Algorithm Engineering:15–29, 1999.

[19] Ning Jing, Yun-Wu Huang, and Elke Rudensteiner. *Hierarchical optimization of optimal path finding for transportation.* In Proc. Of ACM Conference on Information and Knowledge Management, 1996.

[20] Sungwon Jung and Sakti Pramanik. *An efficient path computation model for hierarchically structured topographical road maps.* IEEE Transportations on Knowledge and Data Engineering, VOL 14, NO.5, 2002.

[21] Heechul Lim. *Evaluation of shortest path query algorithm in spatial databases.* a MMath Thesis presented to the University of Waterloo in Computer Science, 2003.

[22] E. Q. V. Martins, M. M. B. Pascoal, and J. L. E. Santos. *An algorithm for ranking loopless paths.* CISUC, Technical Report 99/007, 1999.

[23] E. Q. V. Martins, M. M. B. Pascoal, and J. L. E. Santos. *Deviation algorithms for ranking shortest paths.* International Journal of Foundations of Computer Science, 10(3):247–263, 1999.

[24] E. Q. V. Martins and J. L. E. Santos. *A new shortest paths ranking algorithm.* Investigacao Operacional, 20(1):47–62, 2000.

[25] K. Mehlhorn and M. Ziegelmann. *Resource constrained shortest paths.* Proc. 8th European Symposium on Algorithms (ESA2000) LNCS 1879:326–337, 2000.

[26] Paolo Narvaez, Kai-Yeung Siu, and Hong-Yi Tzeng. *New dynamic SPT algorithm based on a ball-and-string model.* IEEE/ACM Trans. Netw. 9(6):706–718, 2001.

[27] G. Ramalingam and T. Reps. *An incremental algorithm for a generalization of the shortest-path problem.* J. of Algorithms 21:267–305, 1996.

[28] Shashi Shekhar, Andrew Fetterer, and Bjajesh Goyal. *Materialization trade-Offs in hierarchical shortest path algorithms.* SSD:94–111, 1997.

[29] D. R. Shier. *On algorithms for finding the K shortest paths in a network.* Networks 9:195–214, 1979.

[30] Jin Y. Yen. *Finding the K shortest loopless paths in a network.* Management Science 17(11), 1971.

[31] Ning Zhang. *Shortest path queries in very large spatial database.* a MMath Thesis presented to the University of Waterloo in Computer Science, 2001.