Optimization and Evaluation of Shortest Path Queries

Edward P.F. Chan & Heechul Lim
School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
epfchan@uwaterloo.ca
jupithc@yahoo.com
http://sdb.uwaterloo.ca

September 30, 2005

Abstract

We investigate the problem of how to evaluate efficiently a collection of shortest path queries on massive graphs that are too big to fit in the main memory. To evaluate a shortest path query efficiently, we introduce two pruning algorithms. These algorithms differ on the extent of materialization of shortest path cost and on how the search space is pruned. By grouping shortest path queries properly, batch processing improves the performance of shortest path query evaluation. Extensive study is also done on fragment sizes, cache sizes and query types that we show that affect the performance of a disk-based shortest path algorithm. The performance and scalability of proposed techniques are evaluated with large road systems in the Eastern United States. To demonstrate that the proposed disk-based algorithms are viable, we show that their search times are significant better than that of main-memory Dijkstra's algorithm.

1 Introduction

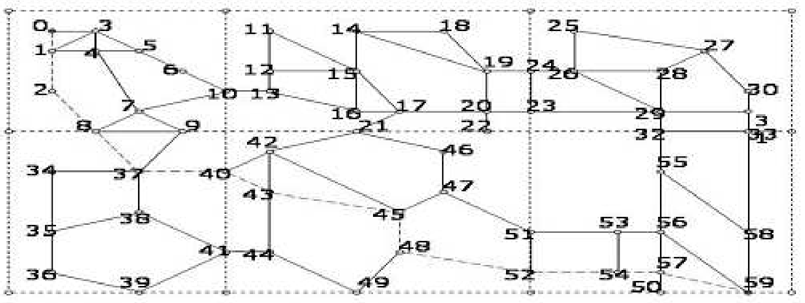
Consider a web-based road information system like Yahoo!Maps or a moving object database [17] in which commuters or users issue queries to find optimal paths from current locations to their destinations. Since road conditions could change over time, the optimal path information needed to be updated constantly as one travels along the route to the destination. As a result, numerous queries are likely issued in a trip by a commuter and thus a large number of queries are expected to be evaluated in such a system. For such a system to be valuable, it is imperative that the route queries are answered efficiently, preferably in real-time.

Among all route queries, *shortest path (SP)* queries are the most popular and have been studied extensively in the literature. In combinatorics, the shortest path problem on general graphs has been well-researched. For example, Dijkstra's algorithm is widely used and is actually very fast when using heap data structures for priority queues [5]. Even faster algorithms are developed, for instance, for graphs that have special constraints on their edge weights [4]. However, one assumption common to all the above algorithms is that the whole graph is stored in the main memory. If the graph is too large, these algorithms cannot handle it. In this paper, we investigate the problem of how to evaluate efficiently a collection of SP queries on massive graphs that are too big to fit in the main memory and must be stored on a disk.

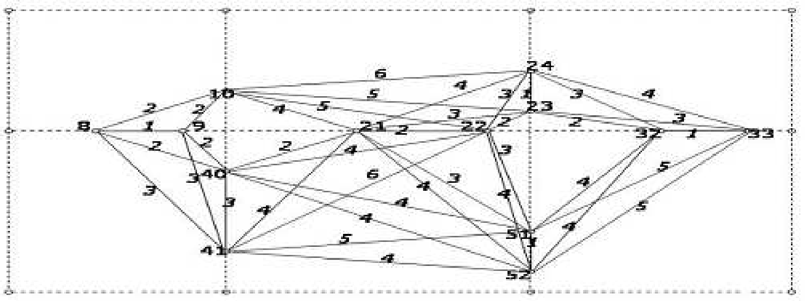
Example 1.1 Suppose we have a road system, abstractly represented as a graph as shown in Figure 1(a). Edges denote street blocks and are bi-directional. To simplify the discussion, let us assume the weight of each street block is 1. For instance, the weight could be interpreted as the time required to travel over a street block. Given such a graph, optimal route queries such as finding an SP from node 0 to node 59 could be posted to the system. Such an SP is indicated as a dotted line in the Figure 1(a). Since a road system in general is too large to be main-memory resident, it can first be partitioned into six subgraphs or fragments as shown in Figure 1(a). For instance, the subgraph containing nodes 0 to 10 forms a fragment. Likewise, the subgraph for nodes 10 to 24 constitutes another fragment. Although we can apply a static SP algorithm like *Dijkstra* on this partition, it may not be effective and could take a long time to process an SP query. For instance, *Dijkstra* requires some auxiliary data associated with each closed node. The amount of data retained during a search could be huge if the number of nodes processed is large. Furthermore, fragments may be read into main memory many times during the processing.

To facilitate the evaluation of route queries, materialization of data is required. A way to speed up the search process is to store the shortest distance between pairs of boundary nodes in a fragment. A node in a partition is *boundary* if it is shared by two or more fragments. For instance, nodes 8, 9 and 10 are boundary nodes. A *super graph* for a partition is a graph with boundary nodes as its nodes and two nodes are connected by an edge precisely when they are in the same fragment. Edges in a super graph are called *skeleton edges.* The weight of a skeleton edge is the shortest distance between two nodes in the fragment involved. The super graph for the partition in Figure 1(a) is shown in Figure 1(b). To evaluate an SP query, an augmented super graph is constructed. An *augmented super graph* for an SP query with source s and destination d is a super graph augmented with the source and destination fragments. Figure 1(c) is an augmented super graph for nodes 0 and 59.

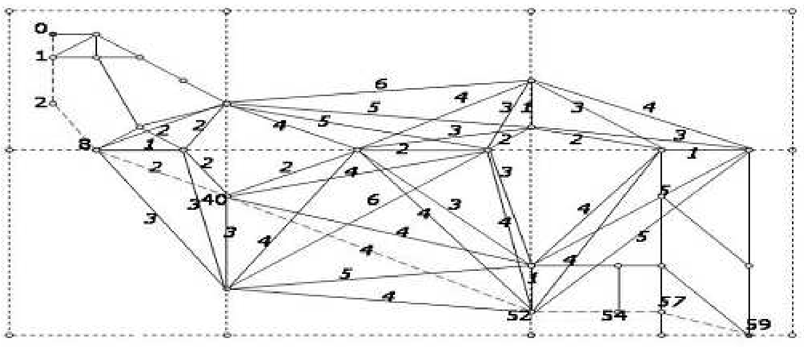
Given an augmented super graph SG for an SP query Q, we can search for an SP P for Q by searching on SG with an algorithm like *Dijkstra.* The dotted line in Figure 1(c) is an SP on the augmented super graph for node 0 to node 59. Once P is found, the actual SP from source to destination can be found by replacing each skeleton edge in P with the corresponding SP in the fragment involved. Such an SP from node 0 to node 59 is shown as the dotted line in Figure 1(a).



(a) A Graph



(b) A Super-graph



(c) An Augmented Super-graph

Figure 1: An Example

Two important properties about a super graph is that, in general, the number of skeleton edges could be comparable to the original graph but the number of nodes is a small fraction of that in the original graph. A super graph can be implicitly represented by storing shortest distances between boundary nodes in a fragment with a matrix. The collection of all matrices for fragments in a partition captures the information in a super graph. Since algorithm like *Dijkstra* processes nodes locally, a matrix can be read into main memory whenever is needed. Thus, the above method of finding an SP is scalable to a very large graph. ■

The method of finding an SP illustrated in the above example is proposed in [3]. It has been shown empirically that, with this method (which we call *DiskSP*), SP's can be found for graphs of millions of nodes and edges, and with the average evaluation time slightly slower than that of *Dijkstra* [3]. However, the search has not been optimized and many nodes in the super graph are searched needlessly. Since scalability and viability are the major problems addressed in [3], issues of optimization and efficient batch evaluation of SP queries are not investigated.

To evaluate an SP query efficiently, we propose and investigate two pruning techniques in this work. To optimize the evaluation of a collection of SP queries, we propose and incorporate batch processing techniques into the SP algorithm. We show that these techniques are effective and their costs are well-justified. Factors such as fragment sizes, cache sizes and query types have a significant influence on the performance of a disk-based SP algorithm. With the proposed SP algorithm, we perform extensive experiments on these factors and their optimality are determined.

In Section 2, we define some basic notation. In Section 3, we survey related work. In Section 4, we describe the proposed algorithms and highlight our contribution. In Section 5, experimental results are presented. Finally, a summary is given in Section 6.

2 Definition and Notation

A network such as a road system is denoted as a graph $G = (V, E, W)$, where V and E are the sets of nodes and edges, respectively, and $W = \{w{:}E \rightarrow \mathcal{R}^{\geq 0} \mid w$ is a function from the set of edges to a set of non-negative real numbers$\}$. For the rest of this paper, we shall use the words "network" and "graph" interchangeably.

Since a graph is too large to be main-memory resident, it is decomposed into smaller pieces called *fragments*. Fragments are stored on disks and each is accessed as a unit during query processing. A *fragment* is a connected sub-graph such that an edge connecting two nodes in a fragment precisely when the two nodes are connected by the same edge in the original graph. A *partition* P of a graph $G = (V,\ E,\ W)$ is a collection of fragments $\{F_1 = (V_1,\ E_1,\ W_1),\ \ldots\ ,\ F_n = (V_n,\ E_n,\ W_n)\}$ such that $\cup_i\ V_i = V$, $\cup_i\ E_i = E$ and $\forall f\ \forall e {\in} E_f$, $w_f(e) = w(e)$. Nodes in a fragment of a partition are divided into two disjoint sets: the *boundary* nodes and the *interior* nodes. A node is a *boundary* node if it belongs to more than one fragment, otherwise it is an *interior* node. An important property of an interior node in a fragment F is that it is adjacent only to interior or boundary nodes of F. It follows that a path connecting an interior node with a node in another fragment must pass through one or more boundary nodes in the fragment of the interior node. On the other hand, a boundary node in a fragment is adjacent to nodes in two or more fragments.

Given any pair of nodes u and v, $SD(u, v)$ is the distance of the shortest path from u to v in G, if u and v are nodes in G and there is a path from u to v, and ∞ otherwise. The *SD* function is said to be *global* if the shortest path is the shortest in the network, while it is *local* if the shortest path is the shortest only in a sub-graph or a fragment. Unless otherwise stated, the *SD* function is global.

To facilitate the query processing, some shortest path information is pre-computed. More specifically, the local shortest distances between all pairs of boundary nodes in each fragment are pre-determined. This information is stored in a *super graph*. The nodes of the super graph are the boundary nodes in the partition. An edge denotes a (local) shortest path between the two nodes in a fragment that contains the two boundary nodes. An edge weight in a super graph is the length of the (local) shortest path between the two nodes, or ∞ if no path connecting them. Note that any pair of boundary nodes could be contained in more than one fragment. In that case, the edge weight in a super graph is the minimum of all (local) shortest distances.

The super graph of a partition can be thought of as a graph consisting of one complete sub-graph (a clique) for each fragment. Formally, a *super graph* $S = (V_s, E_s, W_s)$ of a graph partition $\{F_1 = (V_1, E_1, W_1),\ F_2, \ldots, F_n\}$ has the following properties: $V_s = \{v_b | \exists F_i, v_b \text{ is a boundary node in } F_i\}$, $E_s = \{\langle v_i, v_j\rangle | \exists F_k, v_i \text{ and } v_j \text{ are boundary nodes in } F_k\}$, and $W_s = \{w_s(e_{ij} = \langle v_i,\ v_j\rangle)\ | w_s(e_{ij}) = min\{SD_k(v_i, v_j)|\ 1 \leq k \leq n\}\}$, where SD_k is the local shortest distance from v_i to v_j in fragment F_k, *min* is the minimum function.

3 Related Work

3.1 Previous Work

There are many research efforts reported in the literature on finding the optimal path on graphs that are too large to be main-memory resident. Due to the very large volume of path data, previously suggested transitive closure or graph traversal algorithms [2, 8, 16] are not directly applicable to this problem. Recent research on this problem focus on new data organization techniques for structuring large topographical road data to speed up the computation of the optimal path [1, 14, 10, 7, 3, 11].

All these approaches consist of two distinct phases: *pre-processing* and *querying*. By the assumption that a graph is too large to be main-memory resident, all these methods first convert a graph, by invoking some partitioning algorithm, into a collection of fragments. Each fragment then becomes a unit of transfer between the disk and the main memory. To speed up the search process, either some shortest paths or the cost of some shortest paths are materialized. Each approach differs on how a graph is partitioned into fragments, how these fragments are organized, what path information are materialized, and what search algorithms are employed in the querying phase.

3.1.1 An Early Graph Partitioning Method

Agrawal and Jagadish are among the first to propose the above-mentioned graph partitioning method for finding paths in graphs that are too large to be main-memory resident [1]. A graph is partitioned into fragments and some path information is materialized. An important issue is to reduce the size of the graph needed to be searched. A pruning technique is proposed based on the materialized shortest path cost information. The technique tries to determine if a node needed to be open in algorithms such as A* or Dijkstra's. A node is not considered (open) if the distance from the source to the destination via the node is greater than an estimated distance. While speeding up computation of a shortest path, this technique does not optimize the search process since it unnecessarily searches many nodes in a graph. They also investigate the appropriateness of fragment sizes, materialization of shortest paths, and hierarchical structuring with their method.

3.1.2 Hierarchical Graphs

Another early approach to SP query evaluation that is based on graph partitioning is called *Hierarchical Encoding Path View (HEPV)* and is proposed by Jing et al. [10]. A graph is partitioned using a method called *Spatial Partitioning Cluster (SPC).* Nodes in a fragment are divided into boundary nodes and interior nodes. After partitioning a graph, one locates the boundary nodes for each fragment and computes all-pairs shortest paths among *all* nodes in a fragment, including boundary nodes. A *hierarchical graph* is generated by pushing the boundary nodes up to the next higher level. In each higher level, a super graph is constructed, and is partitioned if it is not small enough to be fitted into the main memory or a page. Edges connect nodes precisely when the two nodes are from the same fragment in the next lower level, and the edge weight is the shortest distance of the two nodes in the fragment in the next lower level. It can be shown easily that the number of levels in a hierarchical graph, if it can ever be generated, is much bigger than 3. Since all-pairs shortest paths and their distances are materialized for nodes in a fragment, the storage and pre-computation time are *enormous* for any reasonable size graph [14].

Given the source s and destination d, the system first looks for the fragments containing the two nodes, say fragments S and D, respectively. S and D are either (a) the same fragment, or (b) they are different. In case (a), the shortest path could be totally in the fragment, or part of it could be in other fragments, thus the path must pass through some boundary nodes of this fragment. In the case (b), the shortest path connecting two different fragments must pass through some boundary nodes in each fragment. Therefore, a common operation of both cases is as follows:

1. Compute the shortest paths from s to all boundary nodes in the S ($s \mapsto BN(S)$) and those from all boundary nodes in D to d ($BN(D) \mapsto d$).

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

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

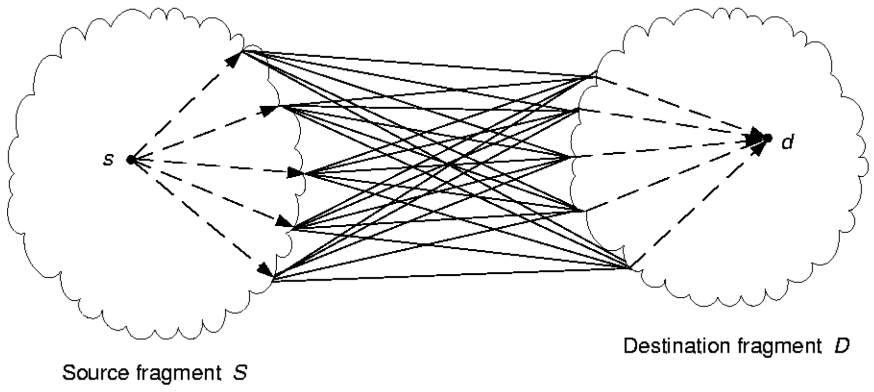


Figure 2: Exhaustive Comparing Algorithm

The shortest path from s to d is the concatenation of a shortest path from s to a boundary node u in S, a shortest path from u to a boundary node v in D, and a shortest path from v to d, i.e., $SP(s,d) = min_{u,v}\{SP(s,u) + SP(u,v) + SP(v,d)\}$.* In Figure 2, the dashed lines in S and D represent the shortest paths from s to boundary nodes in S or from boundary nodes in D to d. The solid lines represent *all possible* shortest paths from boundary nodes in S to the boundary nodes in D. The shortest paths within S and D are easy to acquire, just applying Dijkstra's or other similar algorithm on the fragments. To find the shortest distance between u and v, one must recursively find the shortest path between them one level higher in the hierarchical graph. Since the next level records the shortest path locally, to find the global shortest path for the two boundary nodes requires the shortest path algorithm recursively calls to upper level super graphs. Thus finding such a path between two boundary nodes at the ground level could require a *large* number of invocations of shortest path queries at the upper levels.

Assume that there are k levels in the hierarchy. Then for every query of $SP(u,v)$, we have to go to the top level (level k). Assume that the number of boundary nodes in each fragment is the same, say b. In the ground level (level 1), we need to submit b^2 shortest distance queries to level 2. For every such query, level 2 also needs to submit b^2 shortest distance queries to level 3, and so on, until the top level is reached. Therefore, there are totally $(b^2)^{k-1}$ routing table lookups at the top level. Suppose $b = O(\sqrt{n})$, then the complexity of exhaustive comparing algorithm on multilevel hierarchical architecture is $O(n^{k-1})$. When k is larger than or equal to 3, the asymptotic complexity of the exhaustive comparing algorithm is worse than Dijkstra's algorithm or *DiskSP*.[1] However, if $b = O(n)$, then this method may not be effective.

Shekhar et al. [14] observe that materialization of data is needed to speed up the search process when evaluating an SP query. They investigate the tradeoffs between the storage and computation time, based on a hierarchical structure, with various degrees of materialization. They conclude that materializing the shortest-path-cost view for boundary nodes in fragments provides the best saving in computation time for a given amount of storage. Although the structure assumed is not identical, a similar result is also obtained in [1]. They show that the cost of materializing all shortest paths within a fragment does not justify its benefit in speeding up the search process [1].

Another hierarchical graph model called HiTi is proposed in [11]. In HiTi, a graph is partitioned into fragments as before, except that given any pair of disjoint fragments, the corresponding sets of nodes are disjoint. Boundary nodes are those that connect two fragments. Then a hierarchical graph is generated as in HEPV by pushing the boundary nodes up to the next higher level. In each higher level, two nodes are connected by an edge precisely when they are from the same fragment in the lower level, and the edge weight is the shortest distance of the two nodes in the fragment in the lower level. Since only shortest distances are materialized, the storage cost is much lower than in HEPV. Using an artificially generated grid data set, they show that HiTi outperforms HEPV in terms of update cost and storage overhead. However, in terms of computation cost, HEPV is better.

We implemented the partitioning *SPC* and querying algorithms in HEPV [10], and tested them on road system graphs from Waterloo region. The HEPV method has some termination and performance problems. Firstly, given a graph that is partitioned with the SPC partitioning algorithm, the number of edges in the next higher level in the hierarchical graph may *not* decrease. Thus, the partitioning process may not terminate and the hierarchical graph cannot be generated. Even if the process terminates, by force or otherwise, the number of levels in a hierarchical graph could be numerous and is likely greater than 3. Secondly, even if the partitioning process terminates, the preprocessing time is long and the storage cost is high. Materializing all-pairs shortest pairs in a fragment is an expensive proposition [14]. Lastly, even for those networks whose hierarchical graphs can be generated, the time to find the next hop from a source to a destination with their algorithm is very long. Our experiment shows that, for a graph of several thousand nodes and edges (that is forced to terminate with three levels), the time for finding the next hop in a shortest path ranges from 80 to 3000 times that of Dijkstra's algorithm. Exhaustive search method in a hierarchical graph is not an effective alternative, especially if the graph has more than 2 levels. From these tests, it is clear to us that it is impossible to compare HEPV in our experiment. That is the reason why we did not include it in our comparison in Section 5.

[1]The time complexity of both Dijkstra's and *DiskSP* is $O(n \; log \; n \; + m)$.

3.1.3 Rooted Tree Method

The partitioning algorithm of this approach [7, 18] is an external-memory extension of the Lipton and Tarjan's planar separator algorithm [12]. Therefore, it can only partition planar graphs. Also, the disk-based data structure selected for storing pre-computation information requires a lot of space, thus it may not be suitable for large graphs.

Pre-processing Phase: Constructing Rooted Tree

The rooted tree data structure is a d-nary disk-based tree structure. Each node of the tree corresponds to a fragment and stores its planar graph separator. The children of a node are the connected components separated by its separator, i.e., let G be a parent, S be the separator of G, and $G_1, G_2, \ldots, G_k$ be the resulting partitions of G. If a G_i is not small enough, it is recursively divided and should have its own separator S_i.

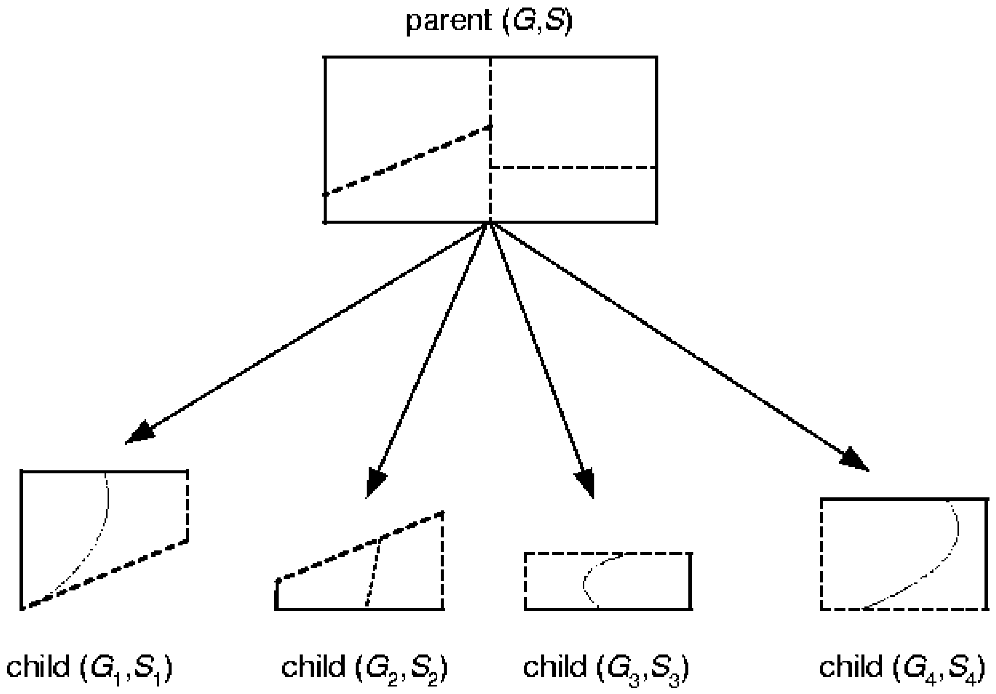


Figure 3: A Rooted Tree (dotted lines are separators)

Let us look at an example. In Figure 3, the parent graph G is first divided into two connected components, then the two components are divided further into four connected components. These four components (G_1, G_2, G_3, G_4) form the next level of (G, S) in the rooted tree. The parent node contains nodes of the separator S for the four connected components. If a G_i is not small enough, it is divided further recursively.

Note that the children of a graph G do not include nodes of the separator S. Therefore, this guarantees that every node in a graph appears in exactly one node in the rooted tree.

Querying Phase

Given a node v, one can find exactly one node (G_k, S_k) in the rooted tree such that v is in S_k (i.e. v is a boundary node of G_k), and v is an interior node of its ancestors $G_i, (i < k)$. Given two nodes u and v, we can find their lowest level common ancestor (G_l, S_l). Define $B(u, v)$ to be the union of the all boundary nodes in their common ancestors, i.e. $B(u, v) = \bigcup S_i, (i \leq l)$. It can be proved that the shortest path between u and v must pass through at least one node in $B(u, v)$.

With this observation, the shortest distance from s to d can be written as : $dist(s, d) = min_{b \in B(s,d)}\{dist(s, b) + dist(b, d)\}$, where $dist$ is the local shortest distance between two nodes. The shortest distance query then can be implemented by an exhaustive comparative algorithm. If the shortest distance from any interior node to any boundary node is already known for each node in the rooted tree, the shortest distance query problem is boiled down to how to efficiently retrieve the shortest distance from one interior node to a boundary node. The thesis [18] discussed the external memory data structure for storing the rooted tree for efficient I/O.

The problem in the above method is that it requires a lot of disk space for storing the shortest distance for the rooted tree and a lot of pre-computation for the shortest distances. Assume that the ground level graph has n nodes. Therefore, there are $O(\sqrt{n})$ boundary nodes on the root if using the Lipton and Tarjan's planar graph separator algorithm. For each interior node, there should be a shortest distance to every boundary node stored in the rooted tree data structure. That is there are $\Theta(n^{3/2})$ shortest distance computations for the pre-computation and $\Theta(n^{3/2})$ shortest distances stored in the tree. For example, in East5 road system used in our experiment, there are about 2.5 millions nodes. Then there will be about 4 billions shortest path computations and about 4 billions shortest distances stored *just* for the boundary nodes in the root of the tree.[2] Thus to materialize every shortest path from an interior node to a boundary node requires a lot of computations and storage.

3.2 A Disk-based Shortest Path Algorithm

A disk-based SP algorithm called *DiskSP* is proposed by us in [3]. With the Tiger/Line file data sets, it has been showed empirically that the algorithm is scalable and its average performance is slightly slower than that of Dijkstra's. The test data sets used represent road networks that contain up to millions of nodes and edges. Since the proposed algorithms in this work are based on algorithm *DiskSP*, we briefly describe the algorithm in this subsection.

[2]With *DiskSP* and with East5 road system, there are about 3 millions shortest path computations and about 3 millions shortest distances stored on a disk.

3.2.1 Pre-Processing Phase

The road systems or graphs used in this work are extracted from the Tiger/Line files [15]. The road network data in a Tiger/Line file cannot be processed efficiently by a route query evaluation algorithm. The following is a description of how we transform the Tiger/Line file data into data that will serve as input in the querying phase.

In a road system in a Tiger/Line file, a node is the intersection of two or more streets while a street block is a part of a street between two adjacent intersections. A set of street blocks are extracted from a road system in a Tiger/Line file. This data is first stored as tables in a relational database. A street block is a polyline denoting the shape of a street block. To facilitate the manipulation of road networks as graphs, street blocks, which are represented as tables in a relational database, are converted into edge objects, which are in fact polylines. Edge objects are stored in a file on a disk. An edge object contains two endpoints, as well as all the interior points in the polyline. Other data associated with a street block (such as its name and weight) are also attached to an edge object. The (default) weight of an edge is the sum of the length of each segment in the polyline. A set of edges together form a very large graph.

The graph in general is too large to be main-memory resident. It is partitioned into a collection of fragments and these fragments are stored in a fragment database. The partitioning algorithm, as proposed in [3], is invoked to generate the fragments. To facilitate the search process, the partitioning algorithm requires the input edge set to be stored in an R-tree or its variant. Thus, a Hilbert R-tree is built on the edges in a graph. After the partitioning, an arbitrary large road network can be partitioned and stored as a collection of fragments in a fragment database. In a Tiger/Line file, some part in a road system is completely disconnected from the rest. With the partitioning algorithm, each of these isolated parts will form a fragment by itself. These isolated fragments will not be used in the experiment. A main-memory version of the whole graph can be generated by reading in each fragment and merging them together. However, this can be done only if the graph is small enough to be main-memory resident.

Conceptually, once a graph has been partitioned, one can apply Dijkstra's algorithm to it, by reading in fragments and their auxiliary data structures from the disk whenever they are needed, and swapping them out when their usefulness expires. However, to minimize computation, memory usage, and I/O accesses, local shortest distances between boundary nodes within fragments are pre-computed. For each fragment in a partition, a *distance matrix* is created to record the distance of a local shortest path from one boundary node to the other. Each matrix is a data structure residing on the disk and they collectively are called a *distance database*. Thus in this method, there are two levels - the *base graph* and the *super graph*. A super graph is *implicitly* represented with a distance database.

3.2.2 Querying Phase: A Disk-based Dijkstra's Algorithm

Once the pre-processing phase is complete, path queries can be posted and evaluated by algorithm *DiskSP*. The algorithm *DiskSP* takes six inputs: the source and destination nodes s and d, two fragments S and D in which s and d are contained respectively, the fragment database $frags$, and the distance database $distDB$. The fragments S and D are called *source* and *destination fragments*, respectively. As with Dijkstra's algorithm, we keep track of information for each node, which includes the shortest distance from source so far, the parent node in the shortest path, and whether it is closed or not. We call these *auxiliary data*. In Dijkstra's algorithm, auxiliary data of nodes are stored in a priority queue ordered by their distances from source.

In our implementation, auxiliary data of nodes in S and D are put in a main-memory priority queue called *SDQueue*. However, the minimum value in this queue alone does not determine the next closed node. Rather the minimum value should be compared with the minimum distance value of all boundary nodes, and the smaller of these two indicates the next closed node. The auxiliary data of boundary nodes are maintained in different priority queues. Since the number of boundary nodes could be huge, their associated auxiliary data must be stored on a disk. To minimize I/O accesses, auxiliary data of boundary nodes are partitioned according to the boundary set in which they belong.

A *boundary set* is the set of boundary nodes shared by two adjacent fragments in a partition. The auxiliary data of nodes in a boundary set are stored in a data structure called *distance vector*. The boundary nodes in a distance vector are organized as a heap and are ordered by their distances from the source. All the distance vectors together form a *distance vector database*. A main-memory heap, called *BNQueue*, contains exactly one entry for each distance vector in the distance vector database. Each entry in *BNQueue* is the minimum-key entry in the corresponding distance vector in the distance vector database. Thus the minimum-key entry in *BNQueue* is the global minimum-key entry for all (open) boundary nodes in the super graph. Unlike a fragment database and a distance database, a distance vector database is a temporary data structure pertaining to a particular query only, and is deleted once the query evaluation is complete. The organization of these heaps is illustrated in Figure 4.

In our implementation, the fragment database, the distance database, and the temporary distance vector database are each stored in a structure called *virtual hashtable*. A virtual hashtable is a hashtable except that elements in it are stored on a disk and at most certain number (*cache size*) of elements are allowed in the main memory. Elements of a fragment database (a distance database and a distance vector database) are fragments (distance matrices and distance vectors, respectively). Elements in a virtual hashtable are swapped in and out of the main memory when the specified cache size is reached. The replacement policy used in a virtual hashtable is the well-known LRU algorithm.

Let us call the graph obtained by merging S, D, and the super graph an *augmented super graph*. Conceptually, Dijkstra's algorithm is applied to an augmented super graph to find the shortest path from the source to the destination. However, since the super graph could be huge, the augmented graph is not constructed explicitly. Instead, only the merged graph of S and D is main-memory resident. During the execution, whenever some part of the super graph is processed, the corresponding distance matrices and

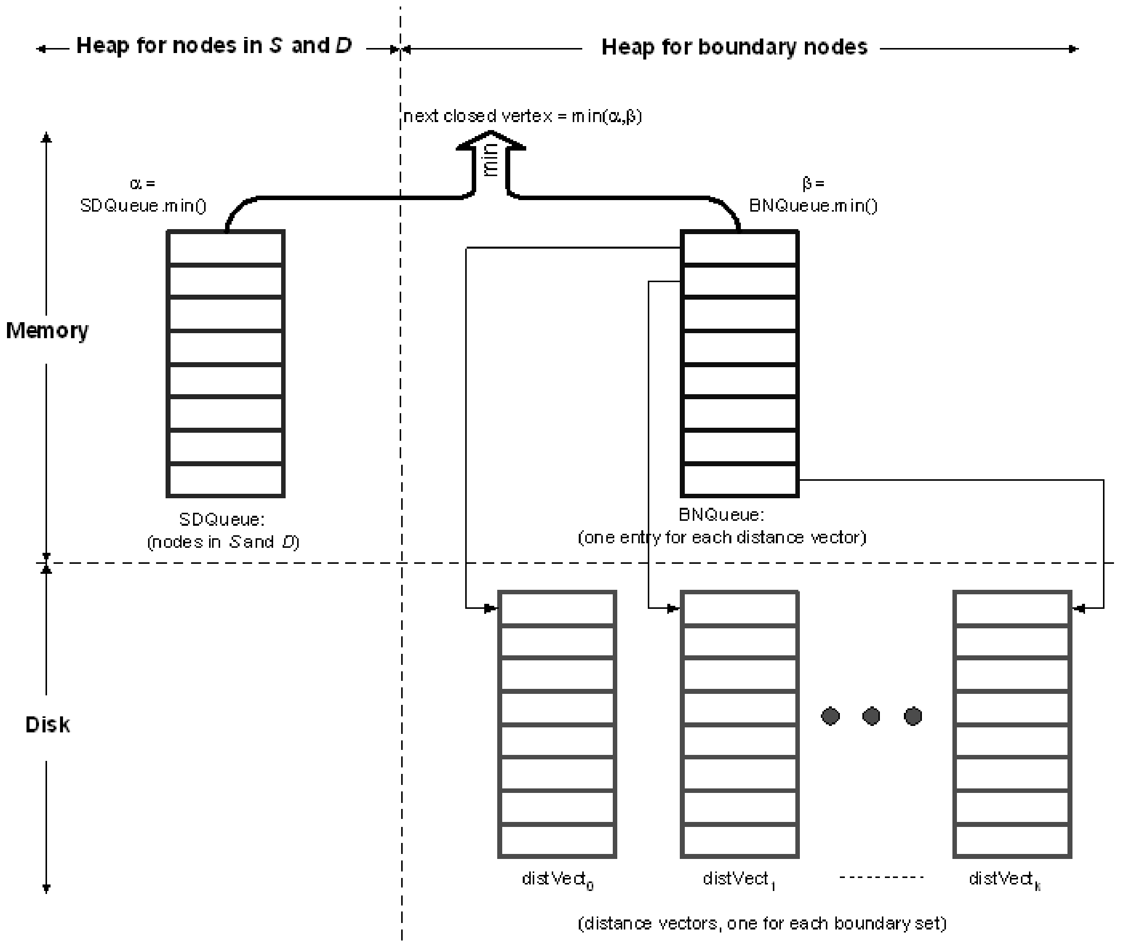

Heap for nodes in S and D
Heap for boundary nodes
next closed vertex = min(α,β)
min
α = SDQueue.min()
β = BNQueue.min()
Memory
Disk
SDQueue: (nodes in S and D)
BNQueue: (one entry for each distance vector)
distVect0
distVect1
distVectk
(distance vectors, one for each boundary set)

Figure 4: Heap Structures For DiskSP

distance vectors are read into the main memory. The shortest path obtained from this augmented graph is not the actual path from the source to the destination, because this path may contain edges in the super graph. An edge $\langle u, v\rangle$ in a super graph represents a local shortest path from u to v in a fragment in which the edge is defined. The last step is to fill-in these edges with the local shortest paths to obtain the complete shortest path from the source to the destination.

Initially, only S and D are in main memory. All nodes (except s) in S, D and all boundary nodes have their distances from s set to ∞. The distance of s is 0. Dijkstra's algorithm is applied to the augmented super graph. In each iteration, an open node with the minimum shortest distance is closed. The closed node could have non-closed interior (in S or D) or boundary nodes as its neighbors. The edge of an adjacent node could be an edge in S, D, or the super graph. During the relaxation process, a non-closed adjacent node is handled in the normal manner. That is, its distance from source is updated and is enqueued into the corresponding priority queue. However, for a non-closed adjacent node which is in the super graph, the corresponding edge weight is stored in a matrix in the distance database. Thus, this process may incur I/O operations if the distance matrix and the corresponding distance vector are not in the cache yet. The relaxation and closing processes keep going until the destination node is closed. After that we get a skeleton shortest path. A *skeleton shortest path*, or simply a *skeleton path*, consists of a sequence of edges in S, D, and the super graph. That is, a skeleton path is a path in the augmented super graph. A skeleton path from node 0 to node 59 is denoted as a dotted line in Figure 1(c). An edge in a super graph is a *skeleton edge*. A skeleton edge represents a shortest path from a boundary node to another boundary node in a fragment to which they belong. The last step of this algorithm is to "fill-in" the skeleton edges. During the"fill-in" step, for each skeleton edge, we apply Dijkstra's algorithm to the corresponding fragment to find the actual path. The path obtained by replacing each skeleton edge with the corresponding actual path is returned as the answer to the query. Details of the algorithm can be found in [3].

4 Our Contribution

Algorithm *DiskSP*, like Dijkstra's algorithm, suffers from the problem that many nodes are needlessly searched and their shortest distances are computed in finding a skeleton path. To reduce the search space, pruning algorithms are introduced in this section. Pruning reduces the super graph needed to be searched, and thus minimizes the number of boundary nodes that are processed. This, as we shall show experimentally, improves the evaluation significantly, by reduction of both I/O and computation.

Algorithm *DiskSP* is designed to process SP queries one by one. It is not intended for an environment in which many queries are executed simultaneously. In such an environment, its performance can further be improved by proper grouping of queries, in both finding the skeleton paths and filling-in the skeleton edges. In the proposed batch processing algorithm, given a set of SP queries, the evaluation is divided into two distinct phases: *skeleton path finding* and *skeleton edge filling*. In the skeleton path finding phase, the queries are first ordered. Then a skeleton path is found, one by one, for each query. A pruning technique could be applied in finding a skeleton path. Once all skeleton paths are computed, the skeleton edge filling phase begins. The skeleton edges in skeleton paths are grouped according to the fragment to which they belong. The actual paths in the same fragment are then found by reading in the corresponding fragment and the shortest paths are computed.

There are factors that could influence the performance of a disk-based SP algorithm. However, these factors are not fully investigated in our work in [3] since optimization is not the focus. The fragment size is important because it affects the search performance as well as the cost of materialized data. In our approach in evaluating an SP query, we first build two shortest path trees, one in the source and one in the destination fragment, before searching is done on the (pruned) super graph. The larger a fragment, the longer it takes to build a shortest path tree. On the other hand, the larger the fragment, the fewer the number of fragments in a partition, and thus the lesser number of boundary sets. Since the performance of a pruning strategy proposed in the work is dependent on the number of boundary sets, the fewer the number of boundary sets, the faster our pruning and thus the faster the search of a skeleton path. In summary, the fragment size cannot be too "large" nor can it be too "small" to obtain an optimized search performance. Furthermore, the fragment size has implication on the cost of materialized data such as a distance database.

In this work, we perform extensive experiments on fragment sizes, cache sizes, and query types that could affect the performance of a disk-based SP algorithm. We show that with proper choices of fragment size and cache sizes, the query evaluation time of these disk-based SP algorithms can be optimized.

A *path query* is denoted by a tuple $\langle src, dst, srcFragId, dstFragId\rangle$, where *src* and *dst* are the source and destination nodes of an SP query, and $srcFragId$ and $dstFragId$ are the ids for the fragments containing the source and destination nodes, respectively. The answer to a path query is a (possibly empty) shortest path in the network from *src* to *dst*. In an environment in which many queries could be posted to a system, there is a queue of size k which denotes k path queries that are batched and executed by the system.

In Section 4.1, pruning algorithms are introduced. In Section 4.2, batch processing of SP queries is discussed. Performance analysis presupposes that data processing has been performed - e.g., see Section 3.2.

4.1 Pruning A Sketch Graph

The pruning algorithms described in this subsection reduce the number of boundary nodes needed to be examined in finding a skeleton path by confirming the search to a sub-graph of the original super graph. We now introduce a concept called *sketch graph*.

A sketch graph captures the connectivity of boundary sets in a partition. The nodes in a sketch graph are boundary sets of the partition, and an edge connects two boundary sets precisely when the two sets are in the same fragment. Formally, a *sketch graph* S

= (V_s, E_s, W_s) of a graph partition $\{F_1, F_2, , \ldots , , F_n\}$ has the following properties: $V_s = \{vs \mid vs$ corresponds to some boundary set in F_i, where $1 \leq\ i \leq\ n\}$. That is, a bijection f: $V_s \rightarrow \bigcup_i BS_i$, where BS_i is the set of boundary sets in the i^{th} fragment. $E_s = \{\langle v_i, v_j\rangle \mid \exists F_k, f(v_i)$ and $f(v_j)$ are in BS_k, where f is the bijection defined on $V_s\}$. The weights of edges in a sketch graph will be defined wherever they are needed.

Given a path query q, the main idea of a pruning algorithm is as follows:

- An upper bound $U(q)$ on the shortest distance from the source to the destination is computed.
- For each boundary set X in the sketch graph, we compute a lower bound $L(q, X)$ on the length of any path from the source to the destination that passes through a node in X. If $L(q, X) > U(q)$, then it is not possible to have a shortest path from the source to the destination via a node in X. Thus, the boundary set X can be pruned.
- The pruned sketch graph, which denotes a sub-graph to be searched, is used to find the skeleton path for q.

The effectiveness of a pruning algorithm depends on how well the upper and lower bounds are computed. To reduce the computation required at query evaluation time, pre-computation and materialization are required. We introduce two pruning algorithms in this section. These two algorithms differ on the accuracy of finding the approximation of the upper and lower bounds, and they also differ on the amount of pre-computation and materialization.

Given two sets of nodes A and B, we define the *minimum* (*maximum*) distance from A to B as the minimum (maximum) of the shortest distances from a node $s \in A$ to a node $t \in B$ in the given network. For the rest of the discussion, the *minimum* (*maximum*) distance from A to B are denoted as *minDist*(A,B) and *maxDist*(A,B), respectively. The function $minDist_{local}$($maxDist_{local}$) is the same as *minDist* (*maxDist*) except that the paths involved are local in a fragment containing both A and B.

4.1.1 Boundary Set Distance Matrix

In this method, to compute $U(q)$ and to determine if a boundary set can be pruned, an additional information on boundary sets is needed. For any pair of boundary sets A and B, we compute the minimum and maximum distances from A to B and from B to A. The minimum and maximum distances for all boundary sets in a partition are stored in a virtual hashtable called *BSDM* (*Boundary Set Distance Matrix*). Elements in *BSDM* are matrices, one for each boundary set. Given two boundary sets A and B, there is a method called *getMin*(A, B)(*getMax*(A,B)) in *BSDM* which returns the minimum (maximum) distance from A to B. *BSDM* is constructed from the super graph during the pre-processing phase.

An Upper Bound $U(q)$

An upper bound on the shortest distance from a source to a destination is computed with the following formula UB_1: min {$maxDist_{local}$({s}, BS_i) + $BSDM.getMin$(BS_i, BS_j) + $maxDist_{local}$(BS_j, {d}) | BS_i and BS_j are boundary sets in the source and destination fragments, respectively}. Likewise, an upper bound can be computed with the following formula UB_2: min {$minDist_{local}$({s}, BS_i) + $BSDM.getMax$(BS_i, BS_j) + $minDist_{local}$(BS_j, {d}) | BS_i and BS_j are boundary sets in the source and destination fragments, respectively}. We choose the minimum of two values computed by the above two formulae as the upper bound $U(q)$. That is, $U(q)$=min{UB_1, UB_2}. $U(q)$ denotes the distance of a path passing some nodes in boundary sets BS_i and BS_j. The number of combinations for each case is $2 \times m \times n$, where m and n are the numbers of the boundary sets in the source and destination fragments, respectively.

We shall prove that $U(q)$ computed is at least equal to, or more than the actual shortest distance from the source to the destination.

Lemma 4.1 *There is a path from the source to the destination whose distance is less than or equal to U(q) computed above.*

Proof It suffices to prove that given any boundary sets BS_i and BS_j from the source and the destination fragments, respectively, the value computed in the formula $U(q)$ is the distance of a path from the source to the destination via nodes in the two boundary sets. Let us focus on the first case UB_1 of the two cases above. The other case can be proven with a similar argument.

Let the upper bound be computed by $maxDist_{local}$({s}, BS_i) + $BSDM.getMin$(BS_i, BS_j) + $maxDist_{local}$(BS_j, {d}). Let v_s and v_d be the nodes in BS_i and BS_j, respectively, such that the distance from v_s to v_d is $BSDM.getMin$(BS_i, BS_j). Consider a shortest path in the source fragment from s to v_s, a shortest path from v_s to v_d, and a shortest path in the destination fragment from v_d to d. The concatenation of these paths is a path from the source to the destination. By definition, $maxSD_{local}$({s}, BS_i)$\geq$$SD$($s$, v_s), where SD is local to the source fragment. Likewise, $maxSD_{local}$(BS_j, {d})$\geq$$SD$($v_d$, d). Thus, the distance of this path from s to v_s to v_d to d is less than or equal to UB_1.

To prove the second case, let the upper bound be computed by $minDist_{local}$({s}, BS_i) + $BSDM.getMax$(BS_i, BS_j) + $minDist_{local}$(BS_j, {d}). Then let v_s and v_d be the nodes in BS_i and BS_j, respectively, such that the distance from s to v_s is $minDist_{local}$({s}, BS_i) and the distance v_d to d is $minDist_{local}$(BS_j, {d}). In that case, the proof follows in a similar argument as in above. ∎

Pruning a Boundary Set

Suppose X, Y and Z are boundary sets and let s and d be two nodes. Assume further that the pair s and X are in the same fragment, and so are d and Z. Define $minDist$(s, X, Y, Z, d) as $minDist_{local}$({s}, X) + $BSDM.getMin$(X, Y) + $BSDM.getMin$(Y, Z) + $minDist_{local}$(Z, {d}). Informally, $minDist$(s, X, Y, Z, d) denotes a lower bound on the distance of a path from node s to d via boundary sets X, Y and Z.

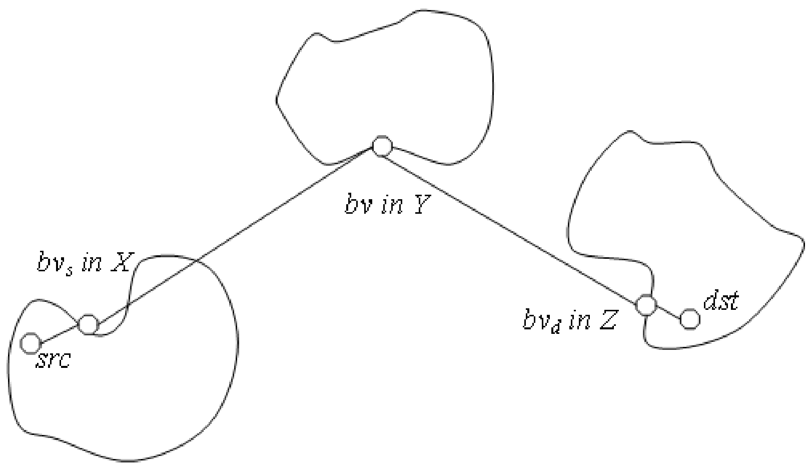


Figure 5: A Path via Boundary Sets X, Y and Z

Lemma 4.2 *Let Y be a boundary set, and let X and Z be boundary sets in source and destination fragments, respectively. Consider a path p from the source src to the destination dst via a boundary node in Y such that the first boundary node and the last boundary node in p are in boundary sets X and Z, respectively. Then the length of p is greater than or equal to minDist(src, X, Y, Z, dst).*

Proof Consider a shortest path from *src* to *dst* passing through some nodes in the boundary sets X, then Y and then Z. As is illustrated in Figure 5, the path p begins at the source *src*, then it passes the first boundary node bv_s in p, then passes boundary node *bv* in Y, to the last boundary bv_d in p, and then to the destination *dst*. The boundary nodes bv_s and bv_d are in X and Z, respectively.

The distance from *src* to bv_s is greater than or equal to $minDist_{local}(\{src\}, X)$. Similarly, the distance from bv_d to *dst* is greater than or equal to $minDist_{local}\ (Z, \{dst\})$. By definition of *BSDM*, the distance $SD(bv_s,\ bv) \geq BSDM.getMin(X,\ Y)$ and $SD\ (bv,\ bv_d) \geq BSDM.getMin(Y,\ Z)$. Thus, the length of the path p is greater than or equal to $minDist(src,\ X,\ Y,\ Z,\ dst)$. ▮

To determine if a boundary set can be pruned in finding a shortest path, we need to consider every path from the source to the destination via this boundary set. By considering all possible combinations of boundary sets in the source and destination fragments, a boundary set can be determined if it is needed in finding a shortest path.

Lemma 4.3 *Consider a path query q= ⟨s, d, sourceFrag, destFrag⟩. Let Y be a boundary set. Define L(q,Y) as the minimum of {minDist(s, X, Y, Z, d) | X and Z*

are boundary sets from sourceFrag and destFrag, respectively}. *If* $L(q, Y) > U(q)$*, then there is no shortest path from* s *to* d *via nodes in* Y.

Proof By Lemma 4.2, any path from s to d via some node in Y must have length greater than $U(q)$. This implies every such path cannot be a shortest path from s to d. ∎

4.1.2 X-Hop Sketch Graphs

Pruning based on *BSDM* takes the advantages of the pre-computed shortest distance information for all-pairs boundary sets. As will be shown later, the pruning algorithm works very well. There are, however, several problems when one applies this algorithm.

The first problem is the calculation time for *BSDM*. To build one entry in a *BSDM*, we have to calculate all shortest paths from nodes in one set to the other. If the graph is huge and each boundary set has a relatively large number of boundary nodes, the calculation time will grow significantly. For example, if each boundary set holds an average of n boundary nodes and the total number of the boundary sets is t, then we need to calculate $n \times t \times t$ *shortest path trees (SPTs)* to fill out all the entries. The other problem is storage space. Since the *BSDM* stores every possible pair of boundary sets, there are t^2 number of entries. Another problem is that edges and their weights in a graph could be updated. Even with a small change in the graph, we must re-build the whole *BSDM*.

An x-hop sketch graph is an alternative solution to *BSDM* that could alleviate the storage and pre-computation problem, but not the dynamic update problem. Unlike that *BSDM* has the shortest distance information on all-pairs boundary sets in a sketch graph, an x-hop sketch graph has the shortest distance information from one boundary set to a limited number of boundary sets, limited by the number x of hops. Therefore, by controlling x, we can adjust the calculation time and storage space for the materialized data.

Given a graph G, the shortest hop from node v_i to node v_j, denoted as $SH_G(v_i, v_j)$, is the distance of the shortest path from v_i to v_j in G, provided that the weights of all the edges in the graph are set to one. Therefore, if $SH_G(v_i, v_j)=h$, then there is a path of h edges from v_i to v_j. Moreover, there is no other path with fewer edges. We said v_j is *h-hops away* from v_i

Given a sketch graph $G=(V,E,W)$, an *x-hop sketch graph* $SG_x = (V_x, E_x, W_x)$ of G, for some positive integer x, has the following properties: $V_x=V$; $E_x = \{e_{ij}= \langle v_i, v_j\rangle \mid SH_G(v_i, v_j)=x\}$; $W_x=\{w_x{:}E_x \rightarrow (\mathcal{R}^{\geq 0}, \mathcal{R}^{\geq 0}) \mid w_x$ is a function from the set of edges to a set of 2-tuples (α, β), where α and β are called α-value and β-value from the boundary set in the head of an edge to the boundary set in the tail$\}$. The α- (β-) value of set A to set B is the $minDist(A, B)$ ($maxDist(A, B)$, respectively). The α-(β-) value is said to be *local* if the corresponding $minDist(A, B)$ ($maxDist(A, B)$) is local. An x-hop sketch graph can be computed in the pre-processing phase. Informally, nodes in an x-hop sketch graph are boundary sets, and an edge from a node u to v if v is x-hops away from u in the sketch graph.

An x-hop sketch graph naturally has more edges than the original sketch graph does, since the number of adjacent nodes of a node grows as x grows. However, with the proper setting of x, the amount of materialized data of an x-hop sketch graph is less than that of *BSDM*. The number of entries of an x-hop sketch graph is $k \times t$, while the one of *BSDM* is always t^2, where t is the number of nodes (boundary sets) and k is the average number of adjacent nodes of a node in an x-hop sketch graph. Since, in general, a node in an x-hop sketch graph does not reach all other nodes, we cannot use it to prune the super graph in the same way as we do in *BSDM*. Instead, we need a more complex scheme to calculate the upper and lower bounds. As we will show later, the effectiveness of this pruning algorithm depends on the choice of x.

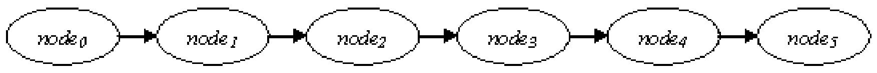
Augmented x-Hop Sketch Graphs

To find the bounds, especially the lower bound, on a shortest path from a node to another, it is imperative to consider all possible paths between them. An x-hop sketch graph will not allow us to find such bounds correctly. However, *augmented x-hop sketch graphs* are sufficient to ensure valid bounds. Therefore one can apply Dijkstra's algorithm to these graphs to compute the bounds.

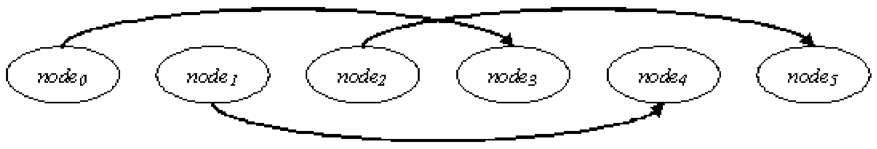
Given a source s and a destination d in fragments S and D, respectively. We construct a graph called a *source-augmented x-hop sketch graph*. Nodes in a source-augmented x-hop sketch graph are boundary sets in the sketch graph augmented with the source and the destination. The set of edges are those in the x-hop sketch graph augmented with the union of the following sets of edges: the set of edges from the source to each source boundary set, the set of edges from each boundary set in the destination fragment to the destination, and the set of edges from a source boundary set to every boundary set in the sketch graph that is less than x-hops away.

Formally, given a positive integer x, a source-augmented x-hop sketch graph $ASG = (V_a, E_a, W_a)$ is obtained from the x-hop sketch graph $SG_x = (V_x, E_x, W_x)$ of a sketch graph G as follows: $V_a = V_x \cup \{s, d\}$; $E_a = E_x \cup E_{sB} \cup E_{Bd} \cup E_{sX}$, where $E_{sB} = \{e_{ij} = \langle s, v_i \rangle \mid v_i$ corresponds to some boundary set in fragment $S\}$, $E_{Bd} = \{e_{ij} = \langle v_i, d \rangle \mid v_i$ corresponds to some boundary set in fragment $D\}$ and $E_{sX} = \{e_{ij} = \langle v_i, v_j \rangle \mid v_i \neq v_j$ and v_i corresponds to some boundary set in S and $SH_G(v_i, v_j) < x\}$. E_{sB} is the set of edges from the source s to boundary sets in the source fragment. Likewise, E_{Bd} is the set of edges from boundary sets in the destination fragment to the destination. Lastly, E_{sX} is the set of edges from a boundary set in the source fragment to a boundary set that is less than x-hops away in the sketch graph. $W_a = \{w_a : E_a \rightarrow (\mathcal{R}^{\geq 0}, \mathcal{R}^{\geq 0}) \mid w_a$ is a function from the set of edges to a set of 2-tuples (α, β), where α and β are the α-value and β-value of the set of nodes at the head of an edge to the set at the tail$\}$. The source and destination nodes in ASG denote singleton sets containing the corresponding nodes in computing the α- and β-values. Moreover, the α- and *beta*-values computed for edges involving the source and destination nodes are *local*.

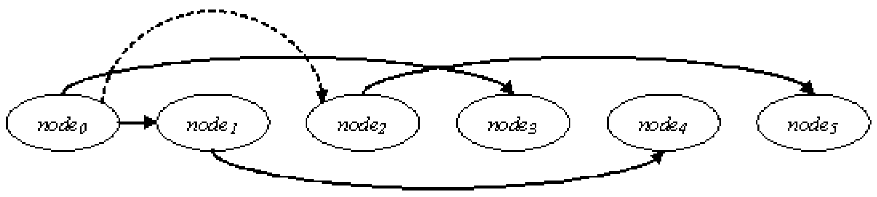
Example 4.1 Let us consider the example in Figure 6. To simplify our discussion, we only show the boundary sets involved, but without the source and the destination



(a) 1-Hop sketch graph



(b) 3-Hop sketch graph



(c) A partial source-augmented 3-Hop sketch graph

Figure 6: An Example of A Partial Source-Augmented x-Hop Sketch Graph.

nodes. Figure 6(a) shows the sketch graph for the set of boundary sets. Figure 6(b) depicts a 3-hop sketch graph SG_3. Suppose that $node_0$ is the only boundary set in the source fragment, Figure 6(c) is a partial source-augmented x-hop sketch graph (but without the source and the destination). The two newly added edges corresponding to the set E_{sX} in the source-augmented x-hop sketch graph. The reason that x-hop sketch graphs are not sufficient in finding valid bounds is because nodes may not be connected. For instance, $node_0$ and $node_5$ are not connected in Figure 6(b). On the other hand, they are connected in an augmented sketch graph, as shown in Figure 6(c). ∎

A *destination-augmented x-hop sketch graph* is the same as a *source-augmented x-hop sketch graph* except the set of edges is $E_a = E_x \cup E_{sB} \cup E_{Bd} \cup E_{Xd}$, where $E_{Xd} = \{e_{ij} = \langle v_i, v_j \rangle \mid v_i \neq v_j$ and v_j corresponds to some boundary set in D and $SH_G(v_i, v_j) < x\}$. An edge from u to v is in E_{Xd} if v is a boundary set in the destination fragment and v is less than x-hops away from the boundary set u. Except edges in E_{sB} and in E_{Bd}, the α- and β-values for edges are global. For edges in E_{sB} and E_{Bd}, the α- and β-values are computed locally *with respect to (wrt)* the source and destination fragments.

An augmented x-hop α- (β-)sketch graph is an augmented x-hop sketch graph with the α-(β-)value as the weight of an edge. An augmented x-hop *dual-weighted* sketch graph is an augmented x-hop sketch graph with the weight of each edge consists of both the α- and β-values.

Properties of an augmented x-Hop Sketch Graph

In this subsection, unless otherwise stated, paths in a graph are paths with both source and destination are *boundary nodes* in the graph. Given a path or a skeleton path p, which contains edges in both the base graph and the super graph. Let $z = bn_j \rightarrow ... \rightarrow bn_k$ be a sub-path of p such that only the first and last nodes of z are boundary nodes. The sub-path z denotes a path in some fragment F_i in the base graph. Let F_i be *$z's$ corresponding* fragment. For each such sub-path z in p, we replace z with $bn_j \xrightarrow{F_i} bn_k$, where F_i is its corresponding fragment. Let the resulting path be $w = bn_0 \xrightarrow{F_{i_1}} bn_1 \xrightarrow{F_{i_2}} bn_2 ... \xrightarrow{F_{i_l}} bn_k$. Given any boundary node bn_j in w, $1 \leq j \leq k$-1, bn_j is a boundary node in the boundary set of fragments F_{i_j} and $F_{i_{j+1}}$. Let bn_0 and bn_k be in some boundary sets BS_0 and BS_k, respectively. Thus each boundary node in w is mapped to precisely one boundary set. The resulting path obtained by replacing each node in w with its unique boundary set is a *boundary set skeleton path (BSSP)* of p, denoted as $BSSP(p)$.

By definition, a BSSP is a path with boundary sets as its nodes. A path in the base graph or a skeleton path can be abstractly denoted by a BSSP. Every edge in a BSSP denotes a sub-path in a skeleton path or a sub-path of a path in the graph.

In a BSSP, some nodes may appear more than once. They are called *repeating* nodes. For example, consider a skeleton path p: $bn_0 \rightarrow bn_1 \rightarrow bn_2 \rightarrow ... \rightarrow bn_n$, where bn_0 and bn_2 are in the same boundary set BS_0. Then, BSSP(p) is $BS_0 \rightarrow BS_1 \rightarrow BS_0 \rightarrow ... \rightarrow BS_n$. The node BS_0 is repeating. A BSSP is said to be *simple* if it has no repeating nodes. A non-simple BSSP can always be converted into a simple BSSP with an order preserving property.

A BSSP q is *order preserving* wrt a BSSP path p, if for any successive boundary sets $BS_i \rightarrow BS_j$ in q, then there is at least one occurrence of BS_i in p, and there is an occurrence of BS_j appears after the last occurrence of BS_i in p.

Lemma 4.4 *Let p: $BS_0 \rightarrow BS_1 \rightarrow BS_2 \rightarrow \ldots \rightarrow BS_n$ be a BSSP of a path in the base graph or of a skeleton path. Then, p can be converted into a simple BSSP q:$BS_0 \rightarrow \ldots \rightarrow BS_n$ such that q is order preserving wrt p.*

Proof The sequence $BS_0 \rightarrow BS_1 \rightarrow BS_2 \rightarrow \ldots \rightarrow BS_n$ is a path in the sketch graph. Let rename this path to t. Repeat the following until there is no repeating boundary set in t: scan t from left to right to find the first occurrence of the first repeating boundary set BS_k; replace the sub-path between the first occurrence and last occurrence of BS_k with BS_k. That is, if $BS_i \rightarrow BS_k$ and $BS_k \rightarrow BS_j$ are the first and last occurrence of BS_k in t, respectively, then the resulting t after the replacement is $\ldots .BS_i \rightarrow BS_k \rightarrow BS_j \ldots$ Note that if $k=0$ or $k=n$, BS_i or BS_j may not exist. It is also worth mentioning that after the replacement, there is no repeating boundary set in the sub-path of t that is up to and including the just-replaced BS_k. We repeat the process of eliminating repeating boundary set until t is simple.

Let the resulting path be q. The path q can easily be proven to be simple in the sketch graph with the first and last boundary nodes BS_0 and BS_n, respectively. A property of q is that, if $BS_i \rightarrow BS_j$ is in q, then BS_i and BS_j must also be in p. Moreover, in p, the successor of the last occurrence of BS_i is BS_j. That is, $BS_i \rightarrow BS_j$ is the last occurrence of the boundary set BS_i in p. The order preserving property thus trivially follows from the transformation. ▮

Algorithm BSSP2XHopSource(p, sg): Transform a simple BSSP p to an order preserving simple path in a source-augmented x-hop sketch graph sg.

Input: A simple BSSP p:$BS_0 \rightarrow BS_1 \rightarrow BS_2 \rightarrow \ldots \rightarrow BS_n$, where BS_0 is a boundary set in the source fragment. A source-augmented x-hop sketch graph sg, for some positive integer x.

Output: An order preserving simple path q wrt p in the source-augmented x-hop sketch graph.

Method:

```
(1) Let $q$ be $p$ and let $LBS$ be the last boundary set $BS_n$ in $q$;
(2) Scan $q$ backward, starting from $LBS$ in $q$, to locate the last
        boundary set $BS$ such that $LBS$ is $x$-hops away from $BS$ in $sg$;
(3) if $BS$ exists then
(4)     replace the sub-path $BS$ to $LBS$ in $q$ with an ($x$-hop) edge $BS \rightarrow LBS$;
(5)     Set $LBS$ in $q$ to $BS$, and go to (2);
(6) else /*$BS$ does not exist.*/
```

```
(7)     If $LBS \neq BS_0$ then
(8)         replace the sub-path $BS_0$ to $LBS$ in $q$ with an edge $BS_0 \rightarrow LBS$;
(9)     end;
(10) end; /*else*/
(11) return $q$;
```

Example 4.2 Let the input to Algorithm *BSSP2XHopSource* be the simple BSSP p and the sketch graph sg in Figure 6(a) and in Figure 6(c), respectively. Then tracing through the algorithm, the first sub-path in p that is replaced by an edge from sg in step (4) is the path $node_2 \rightarrow node_3 \rightarrow node_4 \rightarrow node_5$. The replacing edge in this case is $node_2 \rightarrow node_5$. In the second iteration, LBS is $node_2$ while BS in step (2) does not exist. Therefore in step (8), the sub-path $node_0 \rightarrow node_1 \rightarrow node_2$ is replaced with the edge $node_0 \rightarrow node_2$. Thus the path q returned by the algorithm is $node_0 \rightarrow node_2 \rightarrow node_5$. ▮

Lemma 4.5 *Given a simple BSSP p: $BS_0 \rightarrow BS_1 \rightarrow BS_2 \rightarrow ... \rightarrow BS_n$, where BS_0 is a boundary set in the source fragment. The path p can be mapped into a simple path q in a source-augmented x-hop sketch graph, for any positive integer x, such that q is order preserving wrt p.*

Proof Algorithm *BSSP2XHopSource* consists of two main phases: Steps (2)-(5) and steps (7)-(9). The first phase is an iteration and can be easily verified that it converts a sub-path in p to a path in the source-augmented x-hop sketch graph.

What remains to be shown is that in the second phase, the last sub-path in q that is replaced by an edge in step (8) is valid. Let q' be $BS_0 \rightarrow .. \rightarrow LBS$: the sub-path to the left of LBS in q in the last iteration of the first phase. If, at step (8), $BS_0 = LBS$, then by the assumption in the first phase, the simple path q returned is a simple path in a source-augmented x-hop sketch graph. If $BS_0 \neq LBS$, then by condition in step (2), all boundary sets Y in q' are not x-hops away from LBS. Since every successive boundary set $BS_i \rightarrow BS_j$ in q', BS_j is exactly 1-hops away from BS_i, for all boundary sets Y in q', LBS must be h-hops away from Y, where $h<x$. Since BS_0 is a boundary set in the source fragment, there is an edge in E_{sX} from BS_0 to LBS in a source-augmented x-hop sketch graph. It follows that the simple path q returned by the algorithm is a simple path in a source-augmented x-hop sketch graph, for any positive integer x. The path q is order preserving wrt p follows trivially from the execution of the algorithm. ▮

Let p be a BSSP, p' be the simple BSSP obtained in Lemma 4.4 q be the path returned by algorithm *BSSP2XHopSource* with p' as input. Then, q is said to be the *corresponding* simple path (on the source-augmented x-hop sketch path) of p.

Lemma 4.6 *Let p be a BSSP and let q be its corresponding simple path. Then q is order preserving wrt p.*

Proof Let p' be the simple BSSP obtained in Lemma 4.4. The path p' is order preserving wrt p. Let $BS_i \rightarrow BS_j$ be an edge in q. We want to show that there is an occurrence of boundary set BS_j after the last occurrence of BS_i in p. First, we observe that there are occurrences of BS_i and BS_j in all p, p' and q. Since q is order preserving wrt p', there is a sub-path BS_i→..→BS_j in p'. Since p' is order preserving wrt p, there is an occurrence of BS_j after the last occurrence of BS_i in p. Thus, q is order preserving wrt p. ▮

Lemma 4.7 *Let p be a skeleton path with source s and destination t, where s and t are boundary nodes in boundary sets S and T, respectively, and $S \neq T$. Let BSSP(p) be a BSSP of p. Let q:N_0→N_1→... →N_k be the corresponding simple path of BSSP(p) on a source-augmented x-hop sketch graph, where $k \geq 1$. Then p can be partitioned into k sub-paths SP_1:s=bn_0→... →bn_1, SP_2: bn_1 →... →bn_2,...,SP_k:bn_{k-1}→...→bn_k=t such that bn_i and bn_{i+1} are boundary nodes in boundary sets N_i and N_{i+1}, respectively, for all 0≤i≤k-1.*

Proof To generate the k sub-paths from p, we do the following. For each i =1 to k do: scan p from the beginning until the last occurrence of boundary node bn that belongs to boundary set N_i. Such boundary node bn guarantees to exist since, by Lemma 4.6, q is order preserving wrt BSSP(p). Let the sub-path be SP_i. The first and last nodes of SP_i are boundary nodes in boundary sets N_{i-1} and N_i, respectively. SP_i is the sub-path corresponding to edge N_{i-1}→N_i. Let lbn be the last (boundary) node in SP_i. Remove SP_i from p and insert lbn to the beginning of the updated path p. Go to for loop.

Because of the order preserving property and because q is simple, these k sub-paths guarantee to exist and are non-empty. These k sub-paths form a partition of p. Moreover, for each edge N_i→N_{i+1}, the first and last nodes of the corresponding sub-path are boundary nodes in boundary sets N_i and N_{i+1}, respectively, for all 0≤i≤k-1. ▮

Lemma 4.7 shows that, given any skeleton path p such that the first and last nodes are boundary nodes in different boundary sets, then p can be mapped to a simple k-edges simple path in a source-augmented x-hop sketch graph. An analogous result can be obtained for a skeleton path from a boundary node to another boundary node in the destination fragment. Let p be a BSSP from a boundary node to a boundary node in the destination fragment. Then p can be mapped to a simple path in a destination-augmented x-hop sketch graph as before, except that the following algorithm is applied to transform a simple BSSP to a simple path in the destination-augmented x-hop sketch graph.

Algorithm BSSP2XHopDest(p, sg): Transforms a simple BSSP to an order preserving simple path in a destination-augmented x-hop sketch graph.

***Input*:** A simple BSSP p:BS_0 →BS_1 →BS_2 →... →BS_n, where BS_n is a boundary set in the destination fragment. A destination-augmented x-hop sketch graph sg, for some positive integer x.

***Output*:** An order preserving simple path q wrt p in the destination-augmented x-hop sketch graph.

***Method*:**

```
(1) Let $q$ be $p$ and let $LBS$ be the first node $BS_0$ in $q$;
(2) Scan $q$ forward, starting from $LBS$ in $q$, to locate the last
```

```
        boundary set $BS$ such that $BS$ is $x$-hops away from $LBS$ in $sg$;
(3)  **If** $BS$ exists **then**
(4)      replace the sub-path $LBS$ to $BS$ in $q$ with an ($x$-hop) edge $LBS \rightarrow BS$;
(5)      Set $LBS$ in $q$ to $BS$, and go to (2);
(6)  **else** /*$BS$ does not exist.*/
(7)      **If** $LBS \neq BS_n$ **then**
(8)          replace the sub-path $LBS$ to $BS_n$ in $q$ with an edge $LBS \rightarrow BS_n$;
(9)      **end**;
(10) **end**; /*else*/
(11) **return** $q$;
```

Corollary 4.8 *Let p be a skeleton path with source s and destination t, where s and t are boundary nodes in boundary sets S and T, respectively, and $S \neq T$. Let q:$N_0 \rightarrow N_1 \rightarrow \ldots \rightarrow N_k$ be the corresponding simple path on a destination-augmented x-hop sketch graph, where $k \geq 1$. Then p can be partitioned into k sub-paths SP_1:s=$bn_0 \rightarrow \ldots \rightarrow bn_1$, SP_2: $bn_1 \rightarrow \ldots \rightarrow bn_2$,...,$SP_k$:$bn_{k-1} \rightarrow \ldots \rightarrow bn_k$=t such that bn_i and bn_{i+1} are boundary nodes in boundary sets N_i and N_{i+1}, respectively, for all 0≤i≤k-1.*

Proof It can be proven with a similar argument as in Lemma 4.7. ∎

An Upper Bound $U(q)$

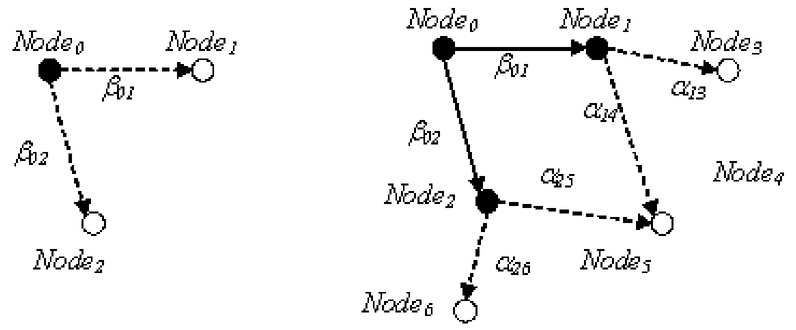
An upper bound on the shortest path from the source to the destination can be obtained by applying Dijkstra's algorithm to a source-augmented x-hop β-sketch graph. However, the bound is sometimes too loose and may not be useful in the pruning process. A better way is applying Dijkstra's algorithm on the same graph with both α- and β-valued edges.

The objective in using the dual-weighted graph is to make the approximation tighter. To accomplish that, we add one more step to the process of Dijkstra's algorithm. In the usual process, we open adjacent nodes of the node which we are going to close. Since each edge of those adjacent nodes has only one value, the distance of each of those open nodes will be $c + o$, where c is the distance of the closed node and o is the weight of the edge from the closed node to the adjacent non-closed node.

Since we use a source-augmented dual-weighted x-hop sketch graph, we have to choose either α- or β-value for each open node to add to the approximation of the closed node. If the approximation of the closed node n is determined by $pc + c$, where pc is the approximation of the predecessor of n, and c, say, is the α-value of the edge from the predecessor to n, then we will choose β-value for the newly open nodes adjacent to n to calculate their approximations. Therefore, we use α- and β-values *alternatingly* when we relax adjacent non-closed nodes, depending on which value the currently closed node uses.

Figure 7 illustrates the process. We determine the distance of $Node_2$ by using the β-value (β_{02}). To determine the distances of its adjacent nodes $Node_5$ and $Node_6$, the α-values (α_{25} or α_{14}, and α_{26}, respectively) are used to calculate their distances. Between

α_{25} or α_{14}, we choose the minimum of (β_{02}+ α_{25}) and (β_{01}+ α_{14}). The bottom line for the algorithm is to choose the minimum of the two approximations: one starting with α-value for the edges of the source node and the other starting with β-value for the edges of the source node.



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

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

Figure 7: Dijkstra's Algorithm on a Dual-Weighted Graph

Lemma 4.9 *The bound obtained with Dijkstra's algorithm on a source-augmented x-hop dual-weighted sketch graph is an upper bound on the shortest distance from the source to the destination.*

Proof Let the path returned be $src \rightarrow BS_1 \rightarrow ... \rightarrow BS_n \rightarrow dst$, where $n \geq 1$. Suppose the first edge weight is a β-value. Then the bound is computed by formula f: $maxDist_{local}$ ($\{src\}$,BS_1)+$minDist$(BS_1, BS_2)+$maxDist$(BS_2, BS_3) +...+ $minDist_{local}$ (or $maxDist_{local}$) (BS_n, $\{dst\}$). For each $minDist$(BS_i, BS_{i+1}), there are two boundary nodes bs_i and bs_{i+1} such that they are in two boundary sets, respectively, and $minDist$(BS_i, BS_{i+1}) = $minDist$($\{bs_i\}$, $\{bs_{i+1}\}$). Since $minDist$ appears alternatingly in f, by replacing each occurrence of BS_i by $\{bs_i\}$ in f, we obtain g: $maxDist_{local}$($\{src\}$,$\{bs_1\}$)+$minDist$($\{bs_1\}$, $\{bs_2\}$)+$maxDist$($\{bs_2\}$, $\{bs_3\}$)+...+$minDist_{local}$(or $maxDist_{local}$)($\{bs_n\}$, $\{dst\}$). Clearly the bound computed by g is less than or equal to that computed by f. Moreover, $src \rightarrow bs_1 \rightarrow ... \rightarrow bs_n \rightarrow dst$ is a skeleton path of a path in the graph. Thus the bound returned is an upper bound on the shortest distance from the source to the destination. Likewise, a similar proof can be constructed if the first edge weight is an α-value. ▮

Pruning Boundary Sets

To find a lower bound from the source to a boundary set Z, we compute an SPT rooted at the source based on a source-augmented x-hop α-sketch graph. The distance of a boundary set Z in the tree denotes the distance of shortest path from the source to any node in Z.

Lemma 4.10 *Let Z be a boundary set. Then the distance computed for node Z by applying Dijkstra's algorithm, starting from the source src, to a source-augmented x-hop α-sketch graph, is less than or equal to the length of any path from src to z, where z is a boundary node in Z.*

Proof It suffices to show that, given a path p in the base graph from *src* to z, there is a path q from *src* to Z in the source-augmented x-hop α-sketch graph such that the length of p is greater than or equal to the length of q.

There are two cases to be considered.

Case 1: There is no boundary node in p except *src* and z. The path p is a local path in the source fragment and Z is a boundary set in the source fragment. Since the α-value from *src* to Z in a source-augmented sketch graph denotes the shortest local distance from *src* to any node in Z, the Lemma holds.

Case 2: There is at least one boundary node in p such that the node is neither *src* nor z. Let p: $src\rightarrow$ ... $\rightarrow z$ be a path in the base graph from *src* to z. Let bn_0 be the first boundary node in p that is neither *src* nor z. The boundary node bn_0 must be in a boundary set BS_0 in the source fragment. Thus p is $src\rightarrow..\rightarrow bn_0\rightarrow...\rightarrow z$, where the sub-path l: $src\rightarrow..\rightarrow bn_0$ is a local path in the source fragment. The length of l is greater than or equal to the α-value from *src* to BS_0 in the augmented sketch graph. Let t be the sub-path $bn_0\rightarrow...\rightarrow z$. If z is in the boundary set BS_0, then clearly the length of path p is greater than or equal to the α-value from *src* to BS_0 in the augmented sketch graph. If z is not in BS_0, by Lemma 4.7, there exists a k-edge simple path w on the source-augmented x-hop sketch graph such that t can be partitioned into k sub-paths, one for each edge in w. Let a sub-path sp_i:$bs_m\rightarrow..\rightarrow bs_n$ in q corresponds to an edge $BS_i\rightarrow BS_j$ in w. By Lemma 4.7, the boundary nodes bs_m and bs_n are in the boundary set BS_i and BS_j, respectively. By definition of an α-sketch graph, the length of sp_i is greater than or equal to the α-value of the edge $BS_i\rightarrow BS_j$ in the source-augmented sketch graph. Thus the length of t is greater than or equal to the length of w. It follows that the length of p is greater than or equal to the length of a simple path from *src* to Z in the source-augmented x-hop α-sketch graph. ∎

Similarly, given the shortest distance of a boundary set Z to the destination, an SPT rooted at destination can be constructed.

Lemma 4.11 *Let Z be a boundary set. Then the distance obtained for destination by applying Dijkstra's algorithm, starting from Z, to the destination-augmented x-hop α-sketch graph, is less than or equal to any path from z to destination, where z is a boundary node in Z.*

Proof It can be proven with Corollary 4.8, and with a similar argument as in Lemma 4.10. ∎

Given a boundary set Z, define $L(s,Z)$ ($L(Z,d)$) as the shortest distance from the source s to Z (from Z to destination d, respectively) in a source-augmented x-hop α-sketch graph (in a destination-augmented x-hop α-sketch graph, respectively).

Lemma 4.12 *Consider a path query $q = \langle s,\ d,\ sourceFrag,\ destFrag\rangle$. Let Y be a boundary set. Define $L(q,Y)$ as $L(s, Y)+L(Y, d)$. Let $U(q)$ be the upper bound computed in Lemma 4.9. If $L(q, Y) > U(q)$, then there is no shortest path from s to d via boundary nodes in Y.*

Proof Suppose there is a shortest path p from s to d via a boundary node y in Y. Let the path be the concatenation of the two sub-paths f:$s \rightarrow ... \rightarrow y$ and l:$y \rightarrow ... \rightarrow d$. By Lemma 4.10, the length of f is greater than or equal to $L(s, Y)$. By Lemma 4.11, the length of l is greater than or equal to $L(Y, d)$. Thus the length of the shortest path p is greater than or equal to $L(q,Y)$, which implies the length of p is greater than $U(q)$. A contradiction to Lemma 4.9. Thus there is no shortest path from s to d via a node in Y. ▮

In using an x-hop sketch graph, the larger x is, the more accurate approximations we can get. However, the larger x is, the more storage and more pre-computation are required. Moreover, the bounds computed may not be as good as in *DSDM*. We will compare the effectiveness of these two approaches in Section 5.

4.1.3 Pruned Sketch Graph

Let us call all those boundary sets Y satisfying the condition in Lemma 4.3 or Lemma 4.12 *pruned* boundary sets. Let us call the super graph, obtained by removing all nodes denoting pruned boundary sets and their incoming and outgoing edges, a *pruned* super graph.

Theorem 4.13 *Given a query $q = \langle s,\ d,\ sourceFrag,\ destFrag\rangle$. Let G_{pruned} be a pruned augmented super graph obtained for q from the augmented super graph G. A shortest path p for q in G iff p is a shortest path in G_{pruned} for q.*

Proof "Only if" A shortest path p from G for q consists of interior nodes in the source and destination fragments and boundary nodes. By Lemmas 4.3 and 4.12, all boundary nodes must be from non-pruned boundary sets. Thus p is a shortest path in G_{pruned} for q.

"If" Let p be a shortest path in G_{pruned} for q. Clearly p is a path in G for q. If p is not the shortest, then there is another path p' in G that is shorter than p. Then in p', there are some node and edge that are not in G_{pruned}. A contradiction to Lemmas 4.3 and 4.12. ▮

4.2 Batch Processing of SP Queries

In an environment in which more than one query is evaluated, the performance of a disk-based SP algorithm can further be improved. The techniques discussed in this subsection primarily reduce the access to the fragment database. For the rest of this paper, a batch SP algorithm is an algorithm incorporating the techniques described in this subsection.

4.2.1 Query Graph

In finding the skeleton path for a path query, only the source and the destination fragments are read into main memory before the skeleton path is found. Thus, the cache size of the fragment database is set to two during the skeleton path finding phase. It is highly desirable to find a sequence of query evaluation so as to minimize accesses to the fragment database. Unfortunately, it has been shown that, given a cache size of two, determining if there is a sequence that results in no fragment is being accessed more than once is an NP-complete problem [13]. Since obtaining an optimal solution is computationally expensive, a heuristic is proposed instead. The algorithm proposed is very simple and fast and its performance is good. The test result of the algorithm will be given in Section 5.

Two queries are said to be in the same *equivalent class* (EC) if the set of source and destination fragments are the same. An EC is denoted by $\{s,\ t\}$, the set of fragments involved. Informally, if two queries are in the same EC, they are executed consecutively so that the utilization of the fragments involved is being maximized. The *query graph* for a set P of path queries is an undirected graph $G{=}\langle N, E\rangle$, where N is the set of fragment id's in P and E is the set of edges for EC's on P. That is, an edge $e{=}\langle a, b\rangle{\in}E$ iff $\{a,b\}$ is an EC on P.

Algorithm *QueryGraph* implements a heuristic that tries to maximize the usage of fragments in the cache. It is assumed in the algorithm that the cache size of fragment database is two, even though it can be generalized to an arbitrary positive integer c. We first define several terms used in the algorithm. A node is *isolated* if it is not connected to any edge in a graph. The node u is called a *terminal* node if the degree of u is one. An edge $e{=}\langle n,\ u\rangle$ is said to be a *dangling* edge incident to n if u is terminal. *Processing an edge* is defined as outputting the edge then removing the edge and removing any isolated nodes from the graph. The edges (EC's) output is the sequence of execution for queries in P.

Algorithm QueryGraph: Find a sequence of EC's based on the graph $G{=}\ \langle E,\ V\rangle$.

***Input*:** A query graph G.

***Output*:** A sequence of edges or EC's, representing the sequence to be executed.

***Method*:**

```
(1)  $cn$ = $null$; /*$cn$ is the current node.*/
(2)  while ($G$ is non-empty) do
(3)      if ($cn == null$) then
(4)          if (there is an edge $e = \langle u, v\rangle$ such that
             $v$ is terminal and the degree of $u$ is either one or two) then $cn = v$;
(5)          else $cn = w$, where $w$ is any node in $G$;
(6)      end; /* if */
(7)      if (there are dangling edges incident to $cn$) then process them one by one;
(8)      if ($cn$ exists) then
(9)          let $e{=}\langle cn, v\rangle$ be an edge incident to $cn$;
```

```
(10)          process $e$;
(11)          $cn = v$, if $v$ exists and $null$ otherwise;
(12)      else $cn = null$;
(13) end; /*while*/
```

At any point of time, *cn* points to the current node being processed. Dangling edges associated with *cn* are all processed consecutively so as to maximize the usage of node pointed at by *cn*. Once the dangling edges are processed, a non-dangling edge is selected to be traversed and processed. The current node is then updated. Every time the **while** loop is executed, an edge is removed. The processing time in the loop is of $O(|V|)$. Thus, the time complexity of the algorithm is $O(|E| * |V|)$.

Example 4.3 Let us consider the query graph in Figure 8. Initially, the current node *cn* is set to node 5 since node 5 is the only node satisfying the condition in statement (4). The edge $\langle 5, 0\rangle$ is processed and is removed. Then the *cn* is set to node 0. Edges $\langle 0, 1\rangle$ and $\langle 1, 2\rangle$ are then processed. After that the current node *cn* is node 2. There are two edges incident to node 2. Arbitrarily select one, say edge $\langle 2, 3\rangle$, and process it. After it is processed, *cn* is set to node 3. This process continues until all edges are finished. The following is a possible sequence output by the algorithm *QueryGraph*: $\langle 5, 0\rangle$,$\langle 0, 1\rangle$,$\langle 1, 2\rangle$,$\langle 2, 3\rangle$,$\langle 3, 8\rangle$,$\langle 8, 9\rangle$,$\langle 8, 6\rangle$,$\langle 6, 7\rangle$,$\langle 6, 4\rangle$,$\langle 4, 2\rangle$. This sequence represents the execution sequence of equivalent classes. If this sequence was executed, only one fragment (fragment 2) is read in from the disk twice. All other fragments are accessed only once. ∎

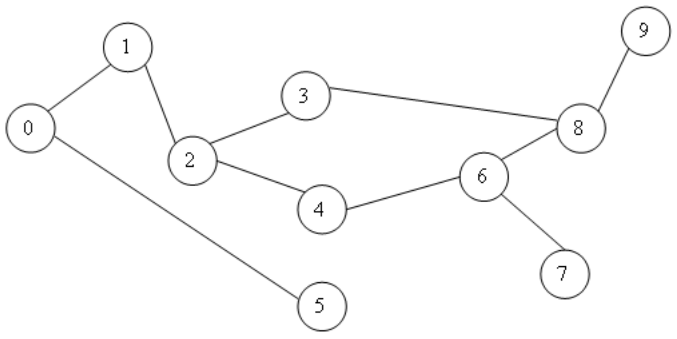


Figure 8: A Query Graph

4.2.2 Skeleton Path Filling

A skeleton edge in a skeleton path represents a shortest path in the fragment to which the skeleton edge belongs. Given a set of skeleton paths, the skeleton edges are grouped according to the fragment they are in. Then in this phase, for each fragment required, it is read into the main memory. The shortest paths are then located for all those skeleton edges defined on the fragment. The actual paths for the skeleton paths are then reconstructed and returned as the answers to the queries.

5 Performance Evaluation

In this section, the performances of proposed algorithms are evaluated. In Section 5.1, statistics are stated on data sets and on the environment in which the tests are conducted. In Section 5.2, the batch algorithm is evaluated. In Section 5.3, the optimality of fragment sizes, cache sizes as well as the optimality of parameters of pruning algorithms are determined. In Section 5.4, results on comparing the proposed algorithms with Dijkstra's and with *DiskSP* are presented.

5.1 Data and Environment

The road system of Connecticut from Tiger/Line file is chosen as our first test case, since the whole system is small enough to be loaded into the main memory and yet large enough to test the proposed disk-based algorithms. The Connecticut road system, when represented as a text file, is about 20MB. It grows to 60MB when it is loaded as a graph into the main memory. It consists of around 190,000 edges and 160,000 nodes. To demonstrate the scalability of the proposed disk-based algorithms, the road system of eastern five states, which is denoted as East5, is used as the second test data set. The East5 is again extracted from Tiger/Line file and is composed of the road systems of Connecticut, Massachusetts, New Jersey, New York, and Pennsylvania. Its consists of more than three million edges and two and half million nodes. The text file occupies about 310MB of storage. This graph cannot be used for Dijkstra's algorithm since it is too big to be main-memory resident, and it is used primarily to test disk-based algorithms. The partitioning algorithm in [3] is used to create a fragment database. Since fragment sizes greatly influence the performance of a disk-based algorithm, databases of different fragment sizes, ranging from 100 nodes to 20,000 nodes per fragment, are used in the experiment.

The system for testing is a Pentium IV 2.56GHz with 1GB of main memory. The hard disk of the system is Ultra ATA-100, with a 7,200 rpm spinning rate. Java is the primary language, and the version is 1.3.1. For Dijkstra's algorithm, we need at least 60MB main memory to load the whole graph of Connecticut. To make a homogeneous environment, for Connecticut and East5 test cases, we set the Java Virtual Machine (JVM) to 128MB and 512MB, respectively.

The statistics of fragment databases are summarized in Table 1 and Table 2. The fragment and distance databases are needed by all disk-based algorithms. In addition, the pruning algorithm in Section 4.1.1 requires the boundary set distance matrix (*BSDM*) as well. For East5, the pruning algorithm with *BSDM* is not evaluated since it takes too much time to generate the *BSDM*.

Table 3 and Table 4 show the file sizes of the x-hop sketch graphs for 1000-node fragment Connecticut and 2500-node fragment East5 databases. As will be shown later, the optimal fragment sizes for Connecticut and East5 data sets are around 1000 and 2500 nodes, respectively. For Connecticut data set, the file size of the graphs tops at the 5-hop sketch graph in the 1000-node fragment, because some of the nodes in the sketch

Frag. Size	Avg Frag. Size (KB)	No. of Frag.	No. of Boundary Sets	No. of Boundary Nodes	Frag. DB Size (MB)	Distance DB Size (MB)	BSDM Size (MB)
100	14	1693	3430	12251	23.4	2.76	185.22
1000	145	138	347	3998	20	2.2	2.02
5000	693	28	66	1649	19.4	1.72	0.09
10000	1379	14	29	1182	19.3	1.72	0.02
15000	1390	10	21	1120	19.3	2.04	0.013

Table 1: Statistics of Fragment Databases for Connecticut Road System

Frag. Size	Avg Frag. Size (KB)	No. of Frag.	No. of Boundary Sets	No. of Boundary Nodes	Frag. DB Size (MB)	Distance DB Size (MB)
1000	148	2163	5141	66479	320	41.5
2500	272	1159	3066	48023	315	40.7
5000	720	432	1148	31311	311	42.3
10000	1330	233	603	22448	310	40.8
20000	2255	137	343	17407	309	40.4

Table 2: Statistics of Fragment Databases for East5 Road System

Frag. Size	x=1	x=2	x=3	x=4	x=5	x=6	x=7	x=8	x=9	x=10
1000	0.395	0.562	0.676	0.745	0.772	0.768	0.735	0.664	0.589	0.519

Table 3: Size (MB) of x-Hop Sketch Graphs for Connecticut Road System

Frag. Size	x=1	x=2	x=3	x=4	x=5	x=6	x=7	x=8	x=9	x=10
2500	4.292	6.079	7.74	9.3	10.75	12.04	13.27	14.38	15.3	16.05

Table 4: Size (MB) of x-Hop Sketch Graphs for East5 Road System

Queue Size (k)	1	10	20	50	100	1000
Cache Utilization	0.0144	0.047	0.12	0.223	0.343	0.471

Table 5: Cache Utilization Comparison with and without QueryGraph

graph do not have 6-hops away nodes; therefore, the file size actually decreases when $x \geq 6$.

The time to build a *BSDM* for a 1000-node fragment Connecticut database is about 8,000 seconds using the system, and its storage cost is about 10% of text data file's size. Generating all x-hop sketch graphs, where $1 \leq x \leq 10$, for a 1000-node fragment, takes about 6,600 seconds and requires total storage of about 30% of the text data file's size. For East5, generating all x-hop sketch graphs, where $1 \leq x \leq 10$, requires about 430,000 seconds. The storage cost of x-hop sketch graphs is again about 30% of the text data file's size.

To investigate whether a query type has any influence on the performance of a disk-based algorithm, we divide queries into three types of ranges: *long*, *medium*, and *short*. Long-range queries are more than 66% of the longest possible distance in the graph, medium-range queries are less than 66% and more than 33%, and short-range queries are less than 33%. We will carry out all the tests according to the differently sized sets of queries. For each query type, 100 queries are randomly generated. Therefore, there are 300 queries in total in a test. Unless otherwise stated, this set of random queries is used in all experiments.

5.2 DiskSP vs Batch DiskSP

In Section 4.2, two batch processing techniques are proposed. These techniques are intended to minimize accesses to the fragment database. In this subsection, we show experimentally that these techniques are effective in reducing the I/O accesses. The improvement on processing time will be demonstrated in Section 5.4.

5.2.1 Query Graph

The fragment database is needed in both finding the skeleton path finding phase and the skeleton path fill-in phase. The usefulness of algorithm *QueryGraph* is for the stage of finding skeleton paths. It attempts to reduce the access to fragment database by re-using the fragments in cache as much as possible. Since at most two fragments are needed in finding a skeleton path, the cache size is set to two. With only two cache entries, the optimal schedule will utilize a maximum of 50% of the cache, if the query graph of the *EC*'s is connected. The reason is that in the optimal schedule, there will be one cache entry for the next query to use. The worst case is 0% of cache utilization.

To evaluate the effectiveness of algorithm *QueryGraph*, 10,000 queries are generated randomly for the 1000-node fragment Connecticut database and are partitioned into groups according to a specific number k. The result of the test is obtained by executing only *QueryGraph* with the pre-sorted *EC*'s of the queries. Therefore, the test is independent of any other phases such as calculating skeleton paths of the queries, or filling the skeleton paths.

Table 5 shows how much the algorithm improves the cache utilization wrt the size of the queries in the queue. The *cache utilization* is calculated by p/q, where p is the number of cache-hits and q the number of total requests for the fragment database.

The case of queue size of one corresponds to the case in which no scheduling is done by algorithm *QueryGraph*. It is clear that more queries in the Queue implies a greater possibility of sharing fragments, and thus a better cache utilization. Even the case of 10 queries in the Queue performs more than three times better than the non-scheduled case. The case of 1,000 queries in the Queue almost reaches the optimal utilization level, which is 50% utilization, whereas the time to schedule is negligible. It takes less than 0.1 seconds for the queue size of 1,000.

5.2.2 Batch Processing Evaluation

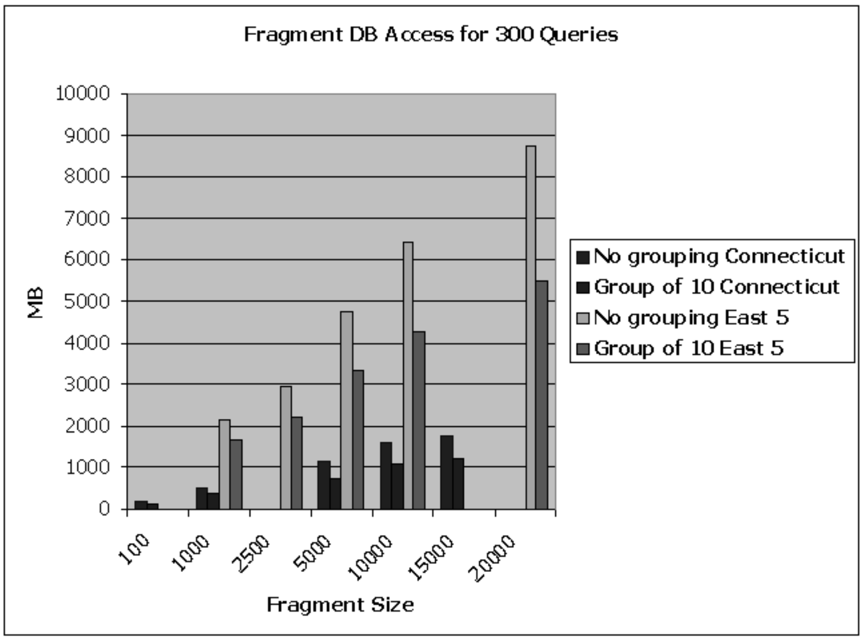


Figure 9: I/O Accesses Comparison with Batch Processing Techniques

The difference between the algorithm *DiskSP* and the batch version lies in how to process multiple queries. The algorithm *DiskSP* executes multiple queries one by one, which means that there is no interruption between queries. Since the batch processing techniques do not have an influence on the performance of access to the distance database, in this subsection, we evaluate the I/O accesses, wrt the fragment database, with queue sizes of 1 and 10. More specifically, in the case of queue size 10, algorithm *QueryGraph* is used to schedule the computation of the 10 skeleton paths.

First, we calculate 300 queries sequentially using algorithm *DiskSP*, and then count the number of requests to the fragment database. We then calculate 300 queries with queue size 10. Since the batch processing techniques introduced are independent of how a skeleton path is computed, the pruning algorithms are not used in this test. The cache sizes for the distance database and for the distance vector database are set to the maximum so that the algorithms are not affected by their cache sizes. The cache size of the fragment database is set to two. We denote these settings, from now on, the *full cache*.

According to Figure 9, the batch version of algorithm *DiskSP*, when compared to algorithm *DiskSP*, reduces accesses to the fragment database by about 20% to 40%. This shows that the ordering by algorithm *QueryGraph* and the batch processing of skeleton path filling contribute to the reduction of I/O accesses towards the fragment database. As will be shown later, fewer fragment accesses during calculations improves the computation time as well.

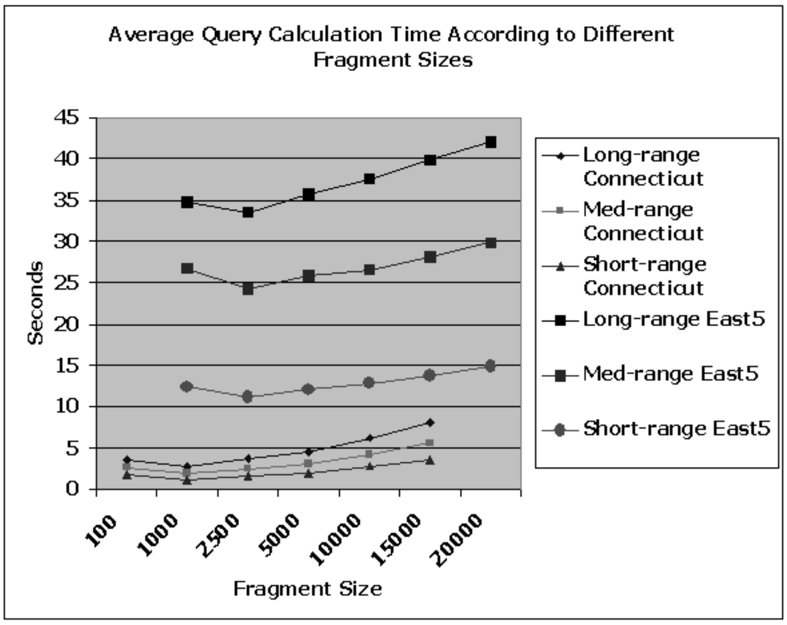


Figure 10: Average Query Evaluation Time for Different Fragment Databases

5.3 Parameter Settings

There are a number of factors that influence the performance of a disk-based SP algorithm. In this section, these factors are investigated and their optimality are determined experimentally.

5.3.1 DiskSP

Since the proposed algorithms are based on algorithm *DiskSP*, we first examine factors that influence its performance. Accesses to the *distance vector database*, the *distance database* and the *fragment database* dominate the I/O cost during an execution of *DiskSP*.

The fragment size has a significant influence on the performance of a disk-based SP algorithm. For both Connecticut and East5 data sets, we test five different fragment sizes. For Connecticut, the sizes are 100, 1000, 5000, 10000, and 15000, whereas for East5, they are 1000, 2500, 5000, 10000, and 20000.

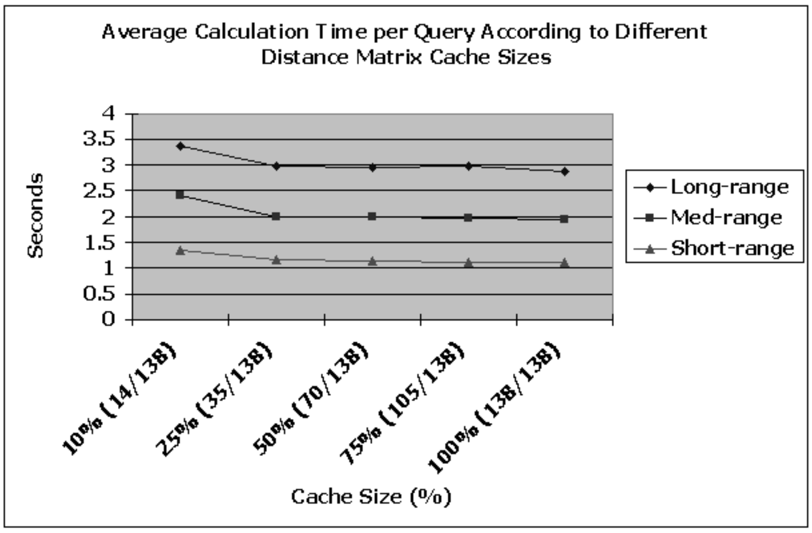


Figure 11: Average Calculation Time for Different Cache Sizes of Connecticut Distance Database

Figure 10 shows the performance differences of algorithm *DiskSP* according to the different fragment sizes for the graphs of Connecticut and East5. For each query type, we process 100 queries, and the time shown in the graph is an average query evaluation time. The cache sizes for the fragment database, for the distance vector database, and for the distance database are set to the full cache. In other words, in determining the optimal fragment size, the I/O activity is isolated.

As shown in the figures, the optimal fragment sizes for Connecticut and East5 data sets are 1000-node and 2500-node fragment databases, respectively. When the fragment size is too small, a super graph is relatively large. Consequently, most of the execution time is spent on searching for the skeleton path in a large augmented super graph. On the other hand, if the fragment size is too "large", building the SPTs in both source and destination fragments takes up most of the execution time.

Having determined the optimal fragment size, we test different cache sizes of distance database. For the 1000-node fragment Connecticut graph and for the 2500-node East5 graph, we test five different cache sizes: 10%, 25%, 50%, 75%, and 100%.

Figure 11 and Figure 12 show the performance differences according to the different cache sizes of distance database. In this experiment, algorithm *DiskSP* is used to find, for each query, the shortest path. The cache sizes of the fragment database and the distance vector database are set to the full cache. For 1000-node fragment Connecticut data set, the optimal value is around 25%. Likewise, the optimal cache size of distance database for the 2500-node fragment East5 data set is around 10%-25%. This shows that as the network grows larger, the optimal cache size for the distance database does

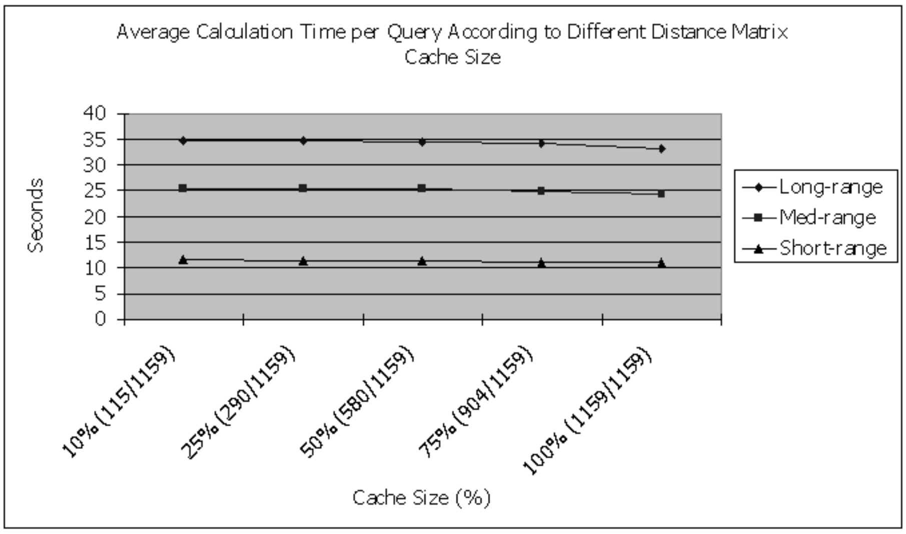


Figure 12: Average Calculation Time for Different Cache Sizes of East5 Distance Database

not increase. For the rest of the experiments, the cache sizes of the distance database for these data sets are set to 25% of their total number of elements.

Finally, the cache size of the distance vector database has a significant impact on the evaluation time. More specifically, if affects the time to find a skeleton path. The cache size for the distance vector database cannot be too low without significantly increasing the skeleton path finding time. For instance, the skeleton path finding time for 1000-node fragment Connecticut database with cache size 1% is about 12 times that of when the cache size is set to 25%. The average calculation time and average I/O accesses on the distance vector database for finding a skeleton path with different cache sizes are shown in Figures 13 and 14, respectively. These results are based on full cache for both fragment and distance databases. It is clear that the larger the cache size gets, the better the performance is. This is particularly true for long and medium queries. Since the distance vector database for a query on the 1000-node fragment Connecticut database and 2500-node fragment East5 are relatively small (around 1.5MB for Connecticut and 25MB for East5), it is worthwhile to set the cache size of distance vector database to the maximum. As we will show later, with pruning, not only the search performance on skeleton paths can be improved significantly, but also its elasticity wrt the cache size of distance vector database is also greatly enhanced.

For the rest of this paper, unless otherwise stated, the two tested databases have the cache sizes of the fragment database, the distance vector database, and the distance database set to their optimal values as determined in this subsection.

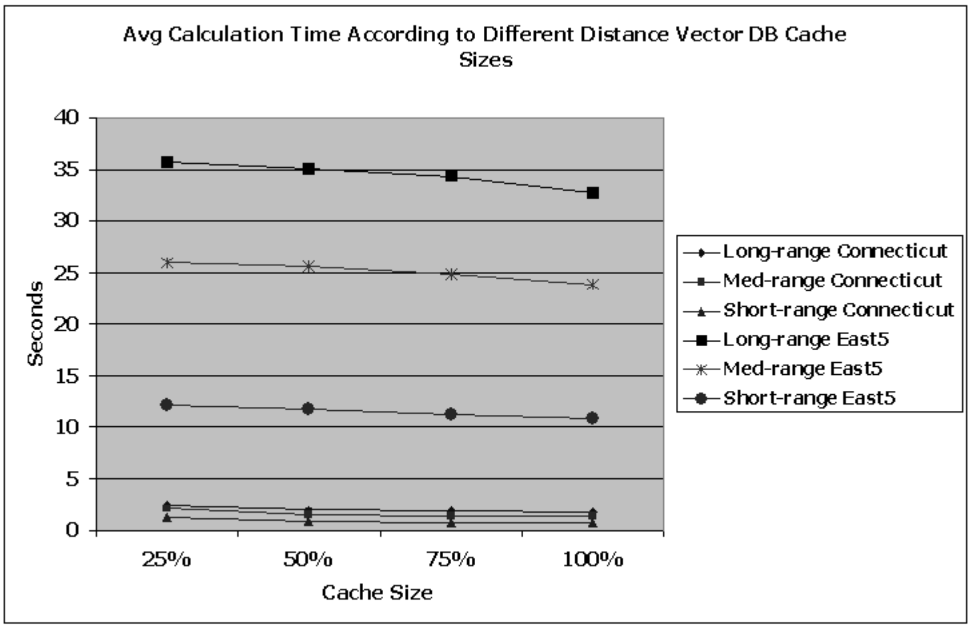


Figure 13: Average Calculation Time Per Query for Different Cache Sizes for Distance Vector Database

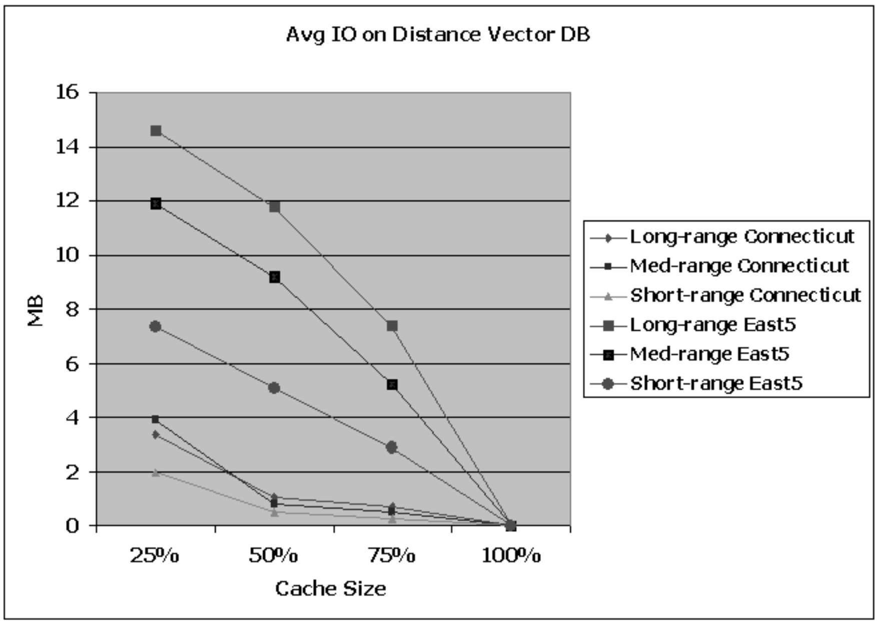


Figure 14: Average I/O Per Query on Distance Vector Database for Different Cache Sizes

5.3.2 Pruning Algorithm with BSDM

The advantage of pruning is to eliminate a number of boundary sets in a sketch graph so that the search space to find a skeleton path is reduced. In a disk-based SP algorithm, there are numerous I/O activities and calculations when the algorithm processes boundary nodes, which makes it important to prune search spaces.

We modified *DiskSP* by incorporating *BSDM* pruning. This algorithm is denoted as $DiskSP_{BSDM}$. Algorithm $DiskSP_{BSDM}$ requires an additional data structure *BSDM*.[3] An element in *BSDM* is a matrix which records the minimum and maximum shortest distances from a boundary set to all other boundary sets. Since the size of a matrix is small (less than 1K in the 1000-node fragment Connecticut database), the cache size of *BSDM* is set to ten. This cache size is large enough to guarantee that each element is accessed at most once during a query evaluation. In other words, the I/O activity of *BSDM* is negligible and will not be considered further for the rest of this paper.

An important metric for measuring the effectiveness of the pruning algorithm is the number of boundary nodes closed during the calculation of skeleton paths. If a node in a sketch graph is pruned, then, from a disk-based SP algorithm's veiwpoint, the boundary set which the node denotes is non-existent. In closing a boundary node, the algorithm has to open and update the distances of adjacent boundary nodes, and the number of the adjacent boundary nodes could be huge.

In the case of the 1000-node fragment Connecticut database, each fragment on average has about 57 boundary nodes, which means every boundary node has on average 56 adjacent nodes. Therefore, in order to close one boundary node, the algorithm has to process a large number of adjacent boundary nodes. Since a boundary set contains a number of boundary nodes, the more nodes (boundary sets) the pruning algorithm eliminates in the sketch graph, the fewer boundary nodes a disk-based SP algorithm accesses during the calculation.

To investigate the effectiveness of pruning based on boundary sets, experiments are conducted on all five Connecticut databases. Figure 15 records how many boundary nodes, on average, are closed in finding a skeleton path of a query. It shows how many boundary nodes we can save from pruning with *BSDM*. For example, the algorithm needs to close about 2100 boundary nodes to calculate a skeleton path of a medium query without pruning in the 1000-node fragment database. On the other hand, the algorithm closes just under 1000 boundary nodes with pruning.

The effectiveness of pruning boundary sets decreases as the fragment size increases. Figure 15 also shows that a pruning algorithm does not work well with larger fragment-sized databases. The reason is that we have a smaller number of boundary sets as we increase the size of each fragment. The difference between the approximations and the actual shortest distance becomes larger as the size of each fragment increases. Therefore, we lose the accuracy of approximations in larger fragment-sized databases. On the other hand, if the fragment size is too "small", say when the fragment size is 100, the number of boundary sets is huge. The benefit of a smaller pruned super graph will be out-weighted by the cost of pruning the boundary sets.

As pointed out in Section 5.3.1, for algorithm *DiskSP*, the cache size of distance vector database is set to full to have the optimal performance. For large networks, it

[3]East5 data set is not used in the test in this subsection since it takes too long to generate the *BSDM*.

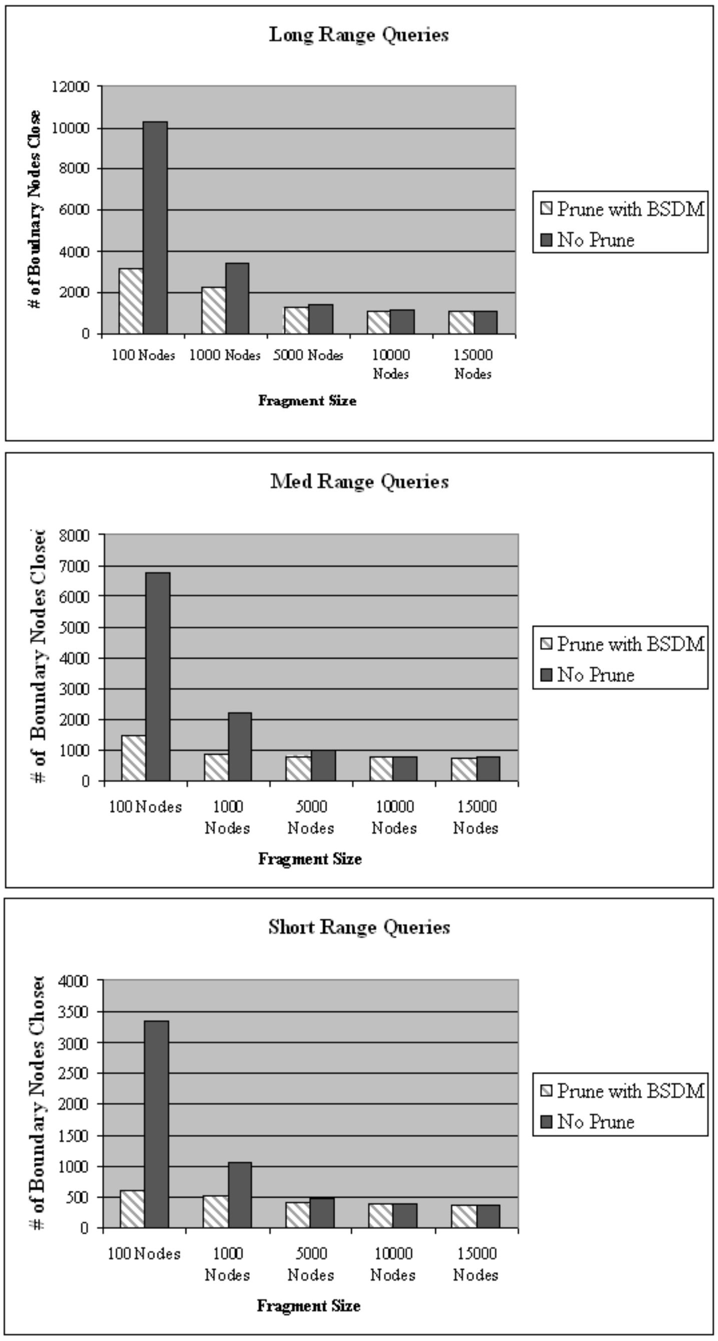


Figure 15: Average Number of Boundary Nodes Closed Per Query With and Without Pruning on Different Fragment-Sized Connecticut Databases

may not be possible or desirable to have a full cache. However, with pruning, the cache size can be reduced significantly and yet still has a better performance than *DiskSP* with full cache. Figures 16 and 17 show the result, by comparing algorithms *DiskSP* and *DiskSP*$_{BSDM}$, on skeleton path finding time and on the distance vector database access. These results are on 1000-node fragment Connecticut database and based on full cache for both fragment and distance databases. From Figure 16, the average skeleton path finding time per query, regardless of query types, with algorithm *DiskSP*$_{BSDM}$ and a cache size of 25%, is significantly shorter than that of *DiskSP* with full cache. For instance, for long-range queries, the skeleton path finding time for *DiskSP*$_{BSDM}$ with cache size 25% and for *DiskSP* with 100% cache are about 1.3 seconds and 1.7 seconds, respectively. Moreover, the influence of the cache size on *DiskSP*$_{BSDM}$ is less significant than that on *DiskSP*. This shows the benefit of pruning on a disk-based SP algorithm.

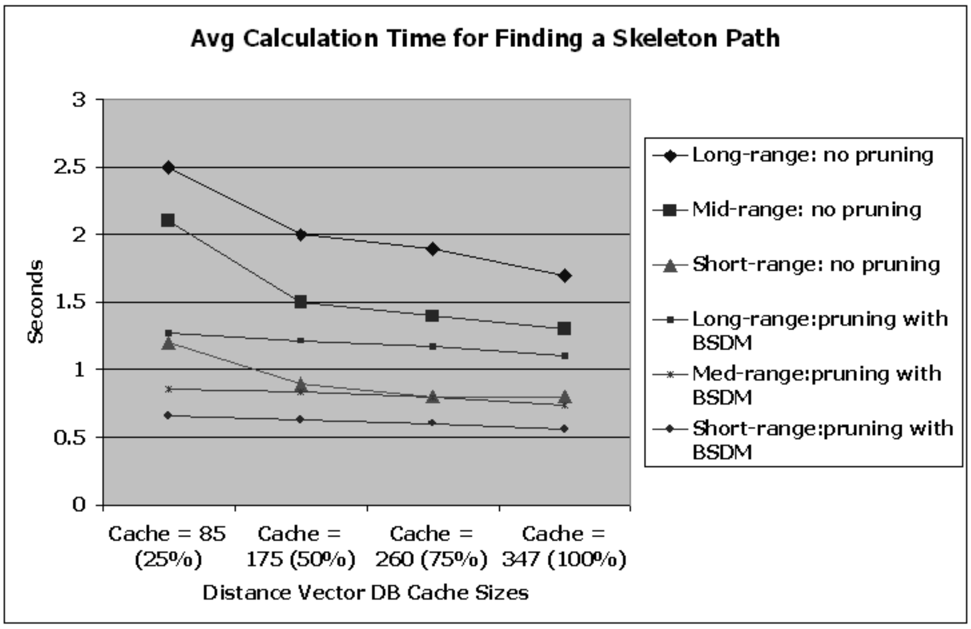


Figure 16: Comparison of Average Calculation Time for Finding a Skeleton Path with Different Cache Sizes of Distance Vector Database

5.3.3 X-Hop Sketch Graphs

Different from *DiskSP*$_{BSDM}$, the disk-based pruning algorithm with an x-hop sketch graph, denoted as *DiskSP*$_{XHOP}$, requires more computation. Given a path query, it needs to compute the upper and lower bounds from the source to all other boundary sets, and from other boundary sets to destination. To find the lower bound from the source to a boundary set, Dijkstra's algorithm is applied to a source-augmented x-hop α-sketch graph. Similarly, we build the SPT rooted at destination with a destination-

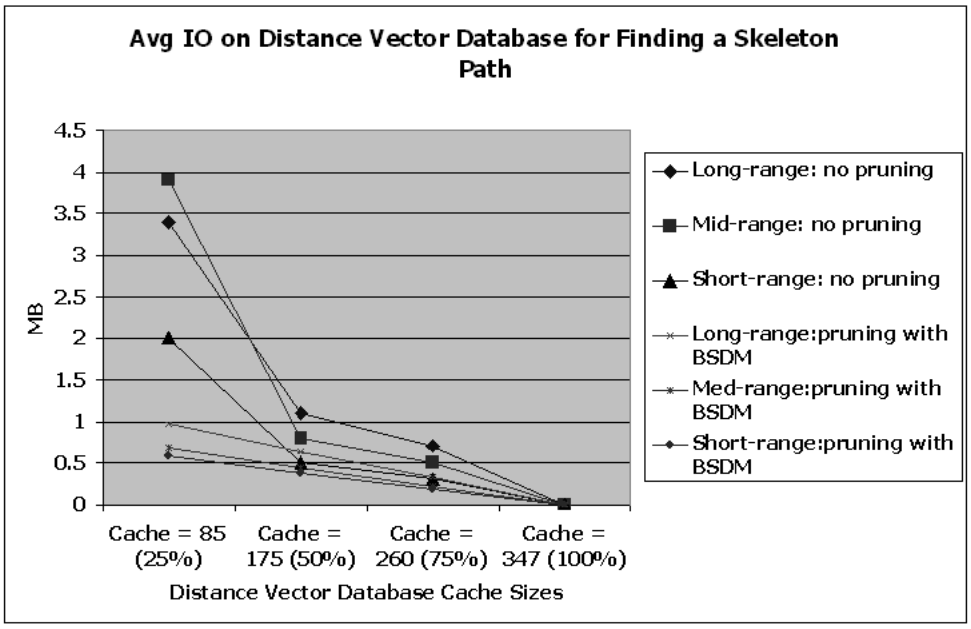


Figure 17: Comparison of Average I/O on the Distance Vector Database Per Query with Different Cache Sizes

augmented x-hop α-sketch graph. Therefore, the pruning procedure takes longer time when compared to $DiskSP_{BSDM}$.

In order to compare the approximations on bounds computed by $DiskSP_{XHOP}$ and by $DiskSP_{BSDM}$, we calculate the α- and β-values of all the boundary set pairs in the sketch graph in 1000-node fragment Connecticut database. Recall that the α- and β-value from set A to set B are the *minDist*(A,B) and *maxDist*(A,B), respectively.

There are 347 boundary sets in the sketch graph, so we have 347^2 cases. We categorize each case by the length computed and divide them into *long*, *medium* and *short* queries.

x	1	3	5	7	9
Long	1.36	1.07	1.035	1.02	1.01
Medium	1.35	1.063	1.026	1.007	1.0006
Short	1.27	1.028	1.0017	1	1

Table 6: Comparison of the β-values

x	1	3	5	7	9
Long	0.24	0.743	0.873	0.93	0.95
Medium	0.265	0.77	0.907	0.96	0.99
Short	0.3	0.862	0.979	0.999	1

Table 7: Comparison of the α-values

For the x-hop sketch graph, we only test the cases of x, where x = 1, 3, 5, 7, and 9. Table 6 shows the comparison for the β-values between the x-hop sketch graph and *BSDM*. The numbers in the table show the average ratio of the β-value computed with the x-hop sketch graph to that computed with *BSDM*. It clearly shows that the values with the x-hop sketch graph become closer to that of $BSDM$ as x increases. Even in the case of long query and $x = 3$, the average value with the x-hop sketch graph is only 7% longer than the one with $BSDM$.

Table 7 shows the comparison for the α-values. The figures in the table also represent the average ratio of the α-value computed with the x-hop sketch graph to that of *BSDM*. Unlike the β-values, the changes become more radical as x increases. In the case of long query and $x = 3$, the average α-value with the x-hop sketch graph is only about 74% of the one with *BSDM*. If we compare the difference of the ratio between the α- and β-values in the same case, we can easily conclude that the pruning algorithm with x-hop sketch graph calculates the β-values better than it calculates the α-values.

To see how the pruning algorithm with an x-hop sketch graph works according to x, we run through ten different x-hop sketch graphs, where $1 \leq x \leq 10$. Figure 18 shows, for all three query types, the average number of closed boundary nodes during the calculations of skeleton paths using these three disk-based SP algorithms. Clearly, *DiskSP* has the largest number of closed boundary nodes while $DiskSP_{BSDM}$ has the least. They represent the two extremes in the graph. The test result shows that the x-hop method converges to *BSDM* as x increases. However, the rate of convergence varies with query types.

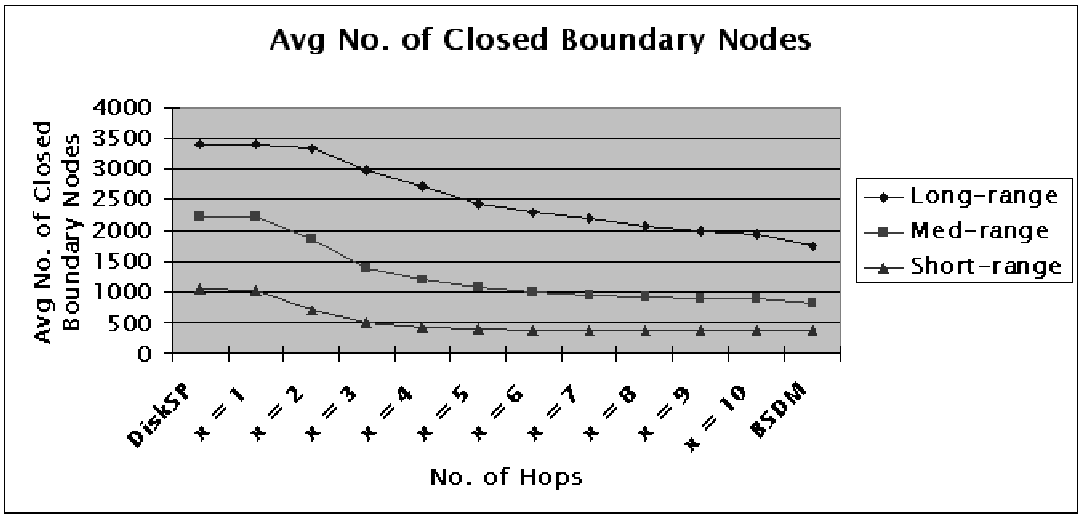


Figure 18: Average Number of Boundary Nodes Closed Comparison on 1000-node fragment Connecticut Database for *DiskSP*, *DiskSP*$_{XHOP}$ and *DiskSP*$_{BSDM}$

Finally, to determine the optimal x for pruning with an x-hop graph, the average query evaluation time by algorithm *DiskSP*$_{XHOP}$ for different query types is computed. Figures 19 and 20 summarize the result on 1000-node fragment Connecticut and 2500-node fragment East5 databases, respectively. From the figures, we observe how the pruning algorithm works with different x's in x-hop sketch graphs. For all three query types, the optimal x's can be found but they are varying from one type of query to another. Therefore, we should choose different x-hop sketch graphs according to the query types. In the subsequent tests, the x-values for Connecticut database are set to 4, 5 and 9 for short, medium and long query types, whereas for East5 the corresponding values are 5, 6 and 8. Although the East5 graph is much larger than the Connecticut graph, their optimal x-values do not differ much. Since the pre-processing and storage cost of algorithm *DiskSP*$_{XHOP}$ are directly proportional to x, this demonstrates that the algorithm works well even for very large graphs.

5.4 Algorithms Evaluation

In this section, we compare algorithms Dijkstra's, *DiskSP*, *DiskSP*$_{BSDM}$, batch *DiskSP*$_{BSDM}$ of 10 queries, *DiskSP*$_{XHOP}$, and batch *DiskSP*$_{XHOP}$ of 10 queries. The batch algorithms accept a queue of k queries. In this test, k is set to 10. During the skeleton path finding phase, k queries are ordered by algorithm *QueryGraph*. The skeleton path for each query is found one by one as in algorithm *DiskSP* but with the corresponding pruning technique incorporated. In the skeleton edge filling phase, skeleton edges for k queries are grouped according to the fragments they reside in. Each relevant fragment is read into the main memory exactly once and the actual paths are computed for the corresponding

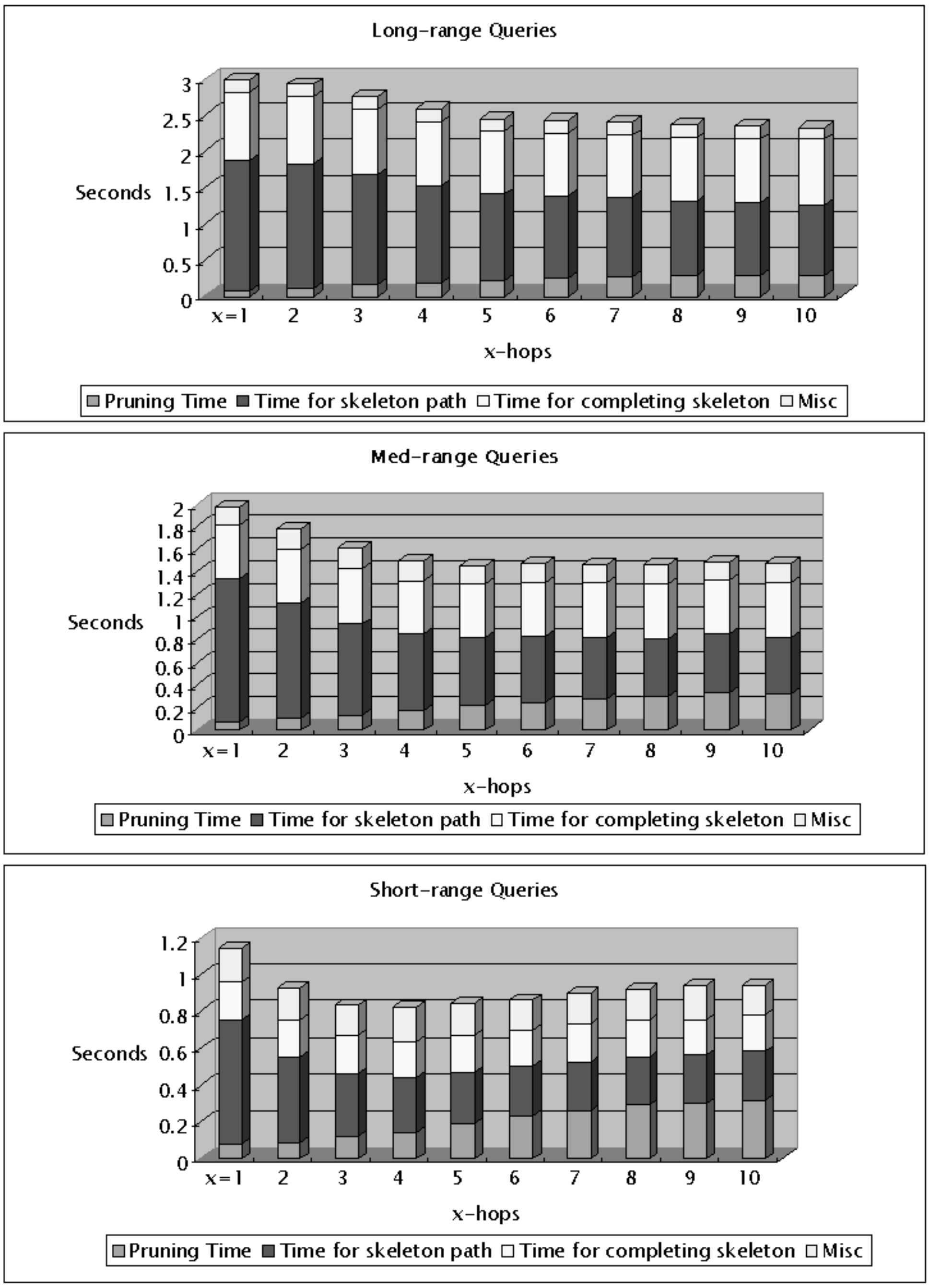


Figure 19: Average Query Evaluation Time on Connecticut Database with $DiskSP_{XHOP}$

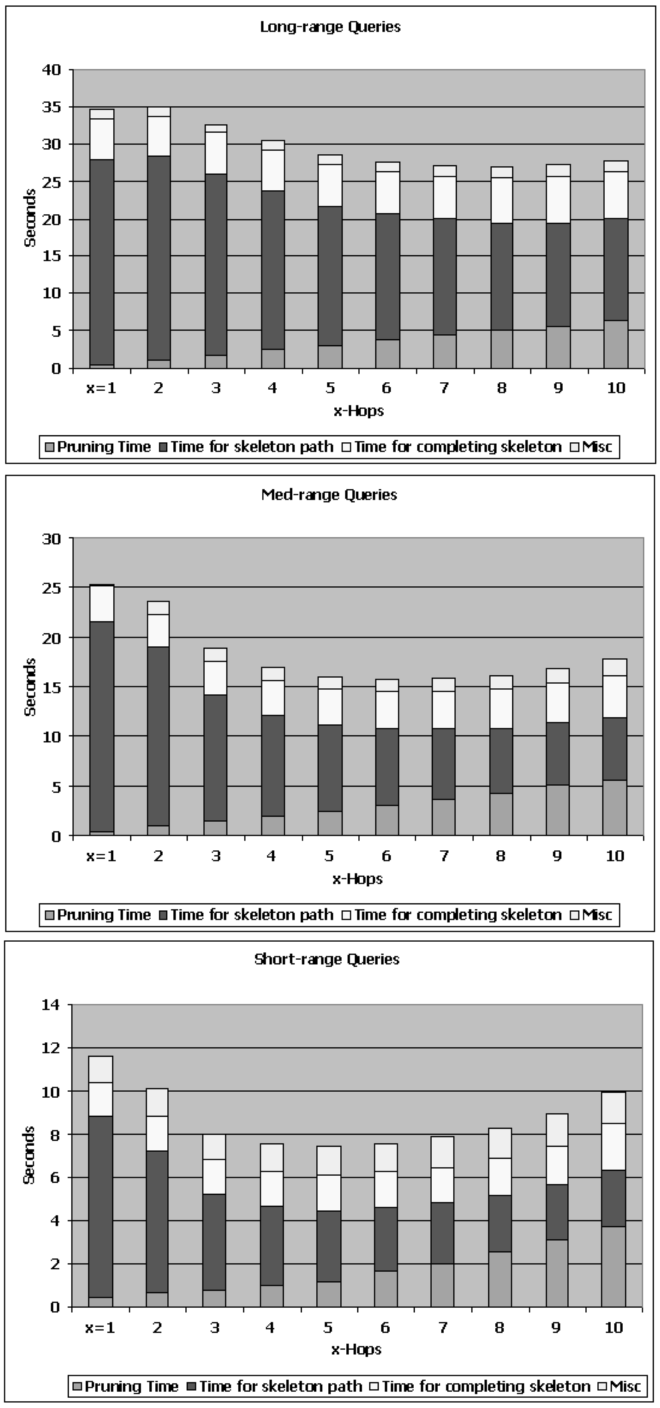


Figure 20: Average Query Evaluation Time on East5 Database with $DiskSP_{XHOP}$

skeleton edges. The databases tested are the 1000-node fragment Connecticut database and the 2500-node fragment East5 database. The cache size of the fragment database, the distance database, and the distance vector database are set to values as shown in Table 8. The x-values of x-hop sketch graphs for long, medium and short queries are set to values as determined in Section 5.3.3.

	Connecticut			East5		
	Distance Vector DB	Distance DB	Fragment DB	Distance Vector DB	Distance DB	Fragment DB
Cache Size	100%	25%	2	100%	25%	2

Table 8: Cache Sizes For Various Databases

To compare these algorithms, we investigate two metrics: the calculation time and I/O activity. The calculation time is, of course, the most important metric, because the whole point of the work is to reduce calculation time. The predominant I/O are on the fragment and distance databases. Since the cache size of the distance vector database is set to full, as is argued in Section 5.3.1, distance vector database accesses are not counted in the overall I/O accesses. Fragment database accesses are essential, since it stores the base graph information. The I/O activity of the distance database is important because distance matrices are the most-used data in calculating a skeleton path. The amount of distance matrix data accessed indicates how well a disk-based SP algorithm performs in evaluating an SP query. For the I/O activity, we do not include the main memory version of Dijkstra's algorithm, because it does not have any I/O activity during calculation.

5.4.1 Execution Time

The first experiment is to evaluate these algorithms' computation time. The computation time is measured as the duration from the beginning of evaluation to the time the answer is returned. The answer to a path query is an actual path on the road system. For each query type, 100 queries are evaluated and their average is determined. The experimental results are summarized in Figures 21 and 23. To compare the query evaluation times between two algorithms A and B, we define *time ratio* of A to B as the query evaluation time by A to that by B. Thus if the ratio is less than one, algorithm A has a better evaluation time.

Figure 21 shows the calculation time for different types of queries on 1000-node fragment Connecticut database. It clearly shows that the main-memory Dijkstra's algorithm performs the worst, and that pruning algorithms in fact reduce the calculation time regardless of query types. The time ratio of a disk-based SP algorithm to Dijkstra's ranges from 30% to 70%. This shows that the proposed disk-based algorithms outperform Dijkstra's significantly.

According to Figure 21, the time ratio of $\textit{DiskSP}_{BSDM}$ to *DiskSP* is about 60% to 70% on the Connecticut database. From Figures 21 and 23, the time ratio of *DiskSP* to $\textit{DiskSP}_{XHOP}$ is about 70% to 80% on the Connecticut database and is about 65% to 80% in East5 database. This shows that the pruning techniques are effective.

Between the two pruning techniques, algorithm *DiskSP*$_{BSDM}$ has 15% to 20% advantage over *DiskSP*$_{XHOP}$ on the Connecticut database. The batch algorithms, when compared with the non-batch version, have an improvement of 3% to 17% on the Connecticut database and an improvement of 0% to 8% on the East5 database. The improvement is the most noticeable when the queries are long. The reason that grouping does not improve much on East5, compared with Connecticut, is due to the larger number of fragments in East5. The number of fragments in 2500-node fragment East5 and 1000-node fragment Connecticut are 1159 and 138, respectively. Given 10 random queries, the chance of having a common fragment between queries in the set is much higher in Connecticut than in East5. Nevertheless, these results show that batch techniques contribute to the improvement in the query evaluation time, independent of the size of the database.

5.4.2 I/O

The results on I/O activities among all *DiskSP* algorithms on the Connecticut database and on East5 are shown in Figure 22 and Figure 23, respectively. Figure 22 illustrates, on average per query, the performance in terms of accessing the distance database and the fragment database. The patterns are the same regardless of the query types. For non-batch version, the accesses to the fragment database are all the same, since for each query, the same skeleton path is returned and thus the fragment database accesses are the same. For the batch algorithms, the same comment holds since the batch processing of k queries returns the same set of skeleton paths, regardless of the pruning algorithms used. Since the same fill-in is performed on the k queries, the accesses to fragment database are the same. However, batch processing reduces the access to the fragment database both in skeleton path finding and in fill-in. With the Connecticut database, the amount of accesses to the fragment database required by batch algorithms is only about 50% of that required by non-batch version in long queries, and 85% in short queries. With the East5 database, the corresponding ratios are 66% in long queries and 95% in short queries.

Between the two pruning algorithms, *DiskSP*$_{BSDM}$ does a better job in pruning the sketch graph and thus requires fewer accesses to the distance database. With the Connecticut database, *DiskSP*$_{XHOP}$ requires 10% to 30% more accesses to the distance database than *DiskSP*$_{BSDM}$ does.

With the Connecticut database, the amount of accesses to the distance database required by *DiskSP* incorporating with pruning techniques is only 25% of that required by *DiskSP* without pruning in short queries, and 65% that in long queries. With the East5 database, the corresponding ratios are 30% in short queries and 55% in long queries. Therefore, the pruning algorithms indeed improve the I/O performance by reducing accesses to the distance database.

Overall, as demonstrated with real-life test data sets, the pruning algorithms and the batch processing techniques proposed reduce the calculation time as well as the I/O activity. We also show that the disk-based algorithms scale well with the increase in the size of a graph.

6 Conclusion and Future Work

We studied the problem of path finding on a massive graph that is too large to be main-memory resident. We proposed several disk-based SP algorithms, and we investigated how a disk-based SP algorithm can be optimized. The performance of these algorithms is heavily influenced by fragment size, cache size of the distance database, and cache size of the distance vector database. We showed experimentally that with proper choices of fragment size and cache size, the performance of a disk-based SP algorithm can be improved noticeably. For 160000-node Connecticut graph, the optimal fragment size is 1000, whereas for 2500000-node East5, the optimal fragment size is 2500. The most important disk-based data structure in the proposed disk-based SP algorithms is the distance database. We showed that for 1000-node fragment Connecticut graph, the optimal cache for distance database is 25%, and for 2500-node fragment East5 graph, the optimal cache is around 10% to 25%. The distance vector database is relatively small, and it is set to full cache in our experiment.

These disk-based SP algorithms are based on the algorithm *DiskSP* [3]. With proper choices of cache sizes and fragment size, we showed that algorithm *DiskSP* outperforms Dijkstra's algorithm by a wide margin. More importantly, it scales well with the increase in the size of the input graph. The performance of *DiskSP* can further be improved. Algorithm *DiskSP* is designed to evaluate an SP query one at a time. Batch processing techniques are incorporated into algorithm *DiskSP* by dividing the evaluation into two steps. The first step is grouping and skeleton path finding, and the second step is skeleton edge filling. In the first step, we sorted the queries so that during the skeleton finding, the fragment database access is minimized. For skeleton path finding, we suggested two pruning algorithms, each of which requires pre-computation to generate materialized data. However, they differ on the effectiveness and the cost of pre-computation. The pruning algorithms return a smaller graph needed to be searched. Consequently, accesses to distance database and distance vector database, as well as the computation time are minimized. In the second step, we grouped a number of skeleton paths computed in the first step and process them together. This results in, at any time, exactly one fragment is processed in the main memory. At the same time, this minimizes the access to the fragment database.

The pruning algorithms significantly contribute to the reduction of calculation time and I/O activities. Even if these techniques need pre-computation and storage, the benefits from the pruning algorithms make them worthwhile. If we deal with a huge graph, such as the East5 road system, we should choose the pruning algorithm using x-hop sketch graphs, because it takes less time and space to build the materialized data. If a graph is small enough, we should choose the pruning algorithm using *BSDM*, because it is more effective and the benefit can be maximized. The cost of batch processing techniques is negligible and these techniques shorten the evaluation time. Therefore, they should be used as long as queries can be processed as a batch. The improvement is most noticeable when a group of queries concentrate on a certain region in the graph.

To demonstrate that the proposed algorithms are viable path finding algorithms, their performance are compared with Dijkstra's. We showed that a disk-based SP algorithm could significantly outperform Dijkstra's algorithm. More specifically, the query evaluation time by a disk-based algorithm could be reduced to as low as 30% of the time required by Dijkstra's. We also showed that the pre-processing and extra storage requirement to achieve such a performance are well-justified.

In summary, we have demonstrated that, with proper choice of fragment size and cache sizes, algorithm *DiskSP* and its variants are effective and scalable for massive graphs. The difference among these algorithms is the cost of generating and maintaining the materialized data. The basic *DiskSP* requires the least cost while algorithm $DiskSP_{BSDM}$ is the most expensive. Algorithm $DiskSP_{XHOP}$, on the other hand, is a compromise between the two extremes.

Pruning with the *BSDM* and with an x-hop sketch graph constitute a trade-off between the efficiency and the cost of the pruning algorithms. Even if we can control the calculation time of x-hop sketch graphs by choosing a proper x, it still takes a long time to build the materialized data if a graph is huge. Another problem with algorithm $DiskSP_{XHOP}$ is that the α-approximations computed is not tight. Further work is needed to improve the approximations.

Throughout the discussion, we assume a graph is static. If the graph is subject to updates, all, or part of the materialized data, such as *BSDM* and x-hop sketch graphs, have to be updated as well, which requires a large amount of processing time. We are currently looking into techniques to effectively maintain materialized data.

We have showed that pruning is essential to the efficient evaluation of an SP query. Thus, having a sketch graph pruning phase before the skeleton path finding phase is desirable in route query evaluation. However, the approach we took in sketch graph pruning is to inspect every node in the sketch graph. Since the running time of the pruning phase is proportional to the number of nodes examined, this method may not be optimized. A more intelligent way of searching nodes in a sketch graph for pruning is needed.

Even after a sketch graph is pruned, the search space can still be reduced further in the skeleton path finding phase. In our proposed algorithm, some boundary nodes in a boundary set in a pruned super graph may still be closed and relaxed even there is no shortest paths from source to destination that can pass through these nodes. A possible direction is to see if pruning of boundary nodes can be carried out dynamically as we search for the skeleton path in the skeleton path finding phase.

In our experiment, we tested the algorithms against Connecticut and East5 data sets. These data sets were chosen because they are real-life road systems. Connecticut is small enough to be main-memory resident, thus it can be used to compare a main-memory and a disk-based SP algorithms. To show the scalability and viability of a disk-based SP algorithm, East5 is selected. East5 road system is too large to be loaded into the main memory of the PC used in the experiment, and, to our best knowledge, is the largest graph that ever tested by a disk-based SP algorithm. However, additional experiments would be needed. Although preliminary tests showed that, once the virtual memory exceeds certain threshold, the performance of *DiskSP* is not affected significantly by the virtual memory size, a more extensive experiment is called for to see if and how the virtual memory affects all data structures and the cache used in the algorithm. The type of graphs may also have influence on the performance of a disk-based SP algorithm. Although road systems we used are real-life data sets, there are other real-life data sets such as graphs representing routing of data in the Internet. It would be interesting to know if and how the performance of the proposed disk-based SP algorithms are affected by other types of graphs.

In our implementation, objects such as fragments and distance matrices are serialized individually to and from the disk. This makes the implementation simply but data are not stored as pages on disk. Consequently, a drawback of our work is that the buffer or cache pool is not page-based, and the I/O performance measure we used is to the number of bytes, not number of pages, read from the disk. A page-based implementation of objects and cache would have been a more realistic choice.

Acknowledgement

We are grateful to the anonymous referees for their comments that made this paper more readable. The authors wish to thank the financial support of the Natural Sciences and Engineering Research Council of Canada.

References

[1] Agrawal, R. and Jagadish, H.V., "Algorithms for Searching Massive Graphs," *IEEE Transactions on Knowledge and Data Engineering, Vol. 6, No. 2,* pp. 225-238, April 1994.

[2] Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," *Proceedings of Sigmod*, 1996, pp. 16-52.

[3] Chan, E.P.F. and Zhang, N., "Finding Shortest Paths in Large Network Systems," *Proceedings of the 9th ACM International Workshop on Advances in Geographic Information Systems* (ACM-GIS 2001), Atlanta, Georgia, pp.160-166, November 2001.

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

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

[6] Goldman, R., Shivakumar, N., Venkatasubramanian, S, and Garcia-Molina H., "Proximity Search in Databases," *Proceedings of the 24th VLDB Conference, New York*, pp. 26-37, 1998.

[7] Hutchinson, D., Maheshwari, A. and Zeh Z., "An External-Memory Data Structure for Shortest Path Queries," *Proceedings of the 5th Annual Combinatorics and Computing Conference (COCOON 99)*, Tokyo, July 26-28, 1999, Lecture Notes in Computer Science, Vol. 1627, Springer Verlag, pp. 51-60.

[8] Ioannidis, Y and Ramakrishnan, R. "Efficient Transitive Closure Algorithms," *Proceedings of 14th VLDB Conference*, pp. 382-394, 1988.

[9] Jagadish, H.V., "A Compression Technique to Materialize Transitive Closure," *ACM Transactions on Database Systems, Vol. 14, No. 4,* pp. 558-598, December 1990.

[10] Jing, N., Huang Y.W. and Rundensteiner, E.A., "Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its performance Evaluation," *IEEE Transactions on Knowledge and Data Engineering, Vol. 10, No. 3,* pp. 1-23, May/June 1998.

[11] Jung, S. and Pramanik, S., "An Efficient Path Computation Model for Hierarchically Structured Topographical Road Maps," *IEEE Transactions on Knowledge and Data Engineering, Vol. 14, No. 5,* pp.1029-1046, September/October 2002.

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

[13] Merrett, T.H., Kambayashi, Y. and Yasuura, H., "Scheduling of Page-Fetches in Join Operation," *Proceedings of the Seventh International Conference on VLDB*, Cannes, France, pp. 488-497, 1981.

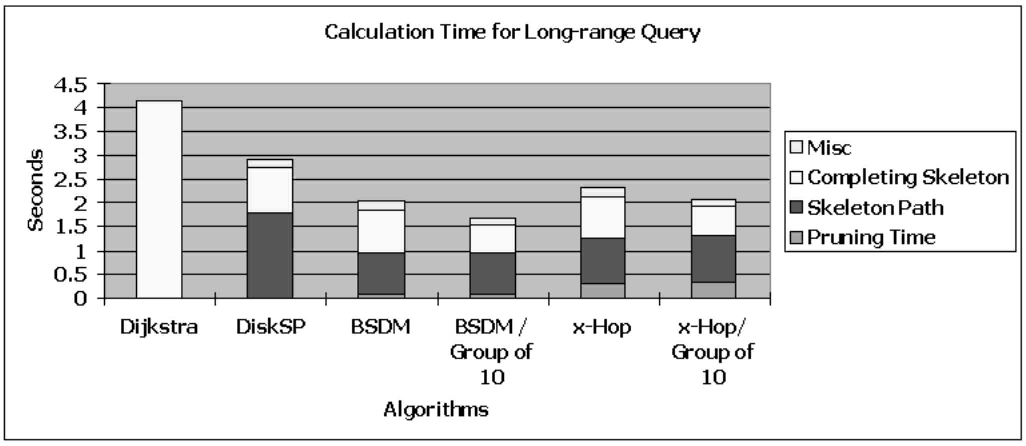
[14] Shekhar, S., Fetterer, A. and Goyal, B., "Materialization Trade-Offs in Hierarchical Shortest Path Algorithms", *Proceedings of the 5th International Symposium on Large Spatial Databases*, pp. 94-111, Lecture Nodes in Computer Science, Vol. 1262, Springer Verlag, 1997.

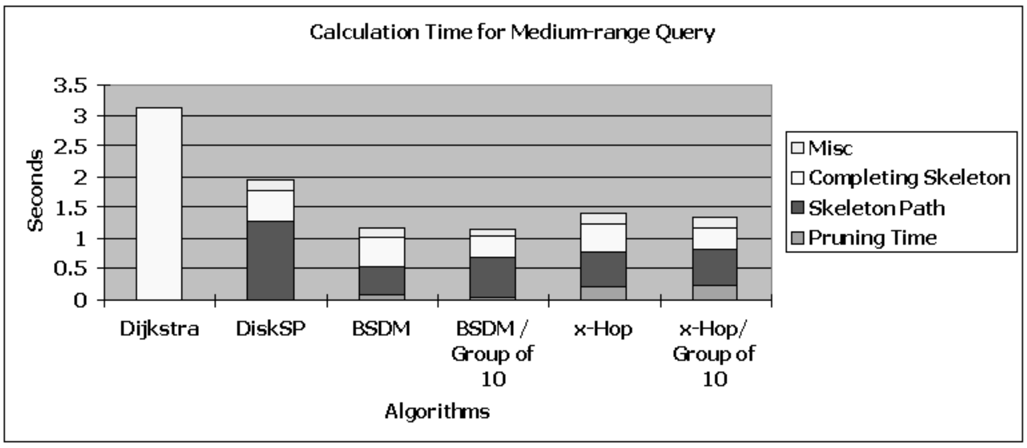
[15] *Tiger/Line Files,* US Department of Commerce Economics and Statistics Administration, Bureau of Census, 1998.

[16] Valduriez, P and Boral, H., "Evaluation of Recursive Queries using Join Indices," *Proceedings of 12th International Conference on VLDB*, pp.403-411, August 1986, Kyoto, Japan.

[17] Vazirgiannis, M and Wolfson, O., "A Spatialtemporal Model and Language for Moving Objects on Road Networks," *Proceedings of the 7th International Symposium on Spatial and Temporal Databases (STTD 2001)*, pp. 20-35, L.A., CA, July 2001.

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





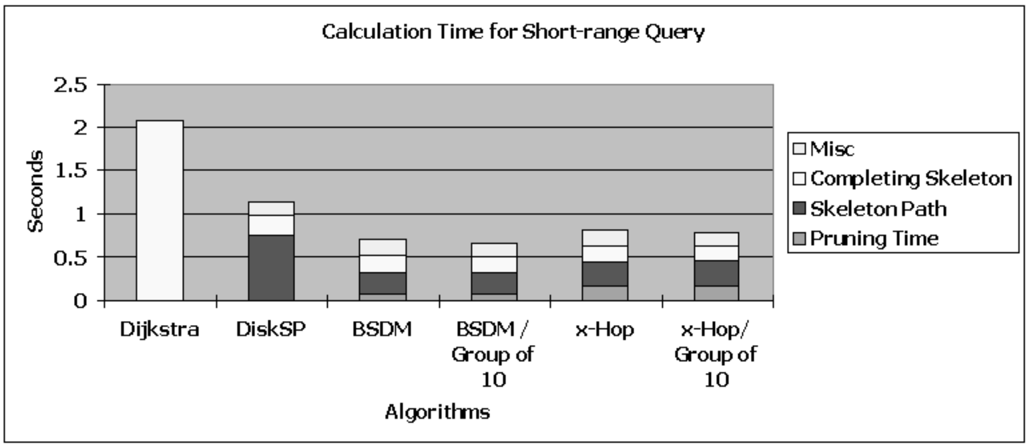


Figure 21: Average Calculation Time per Query on Connecticut Database for Different Algorithms

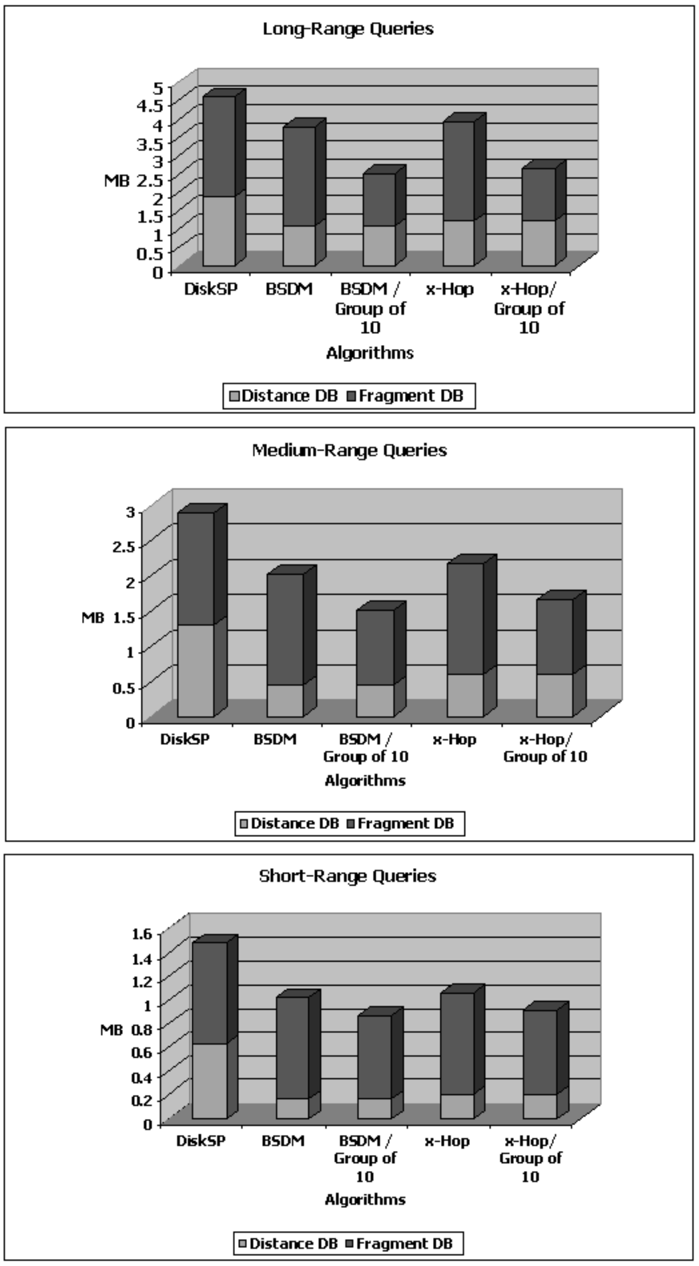


Figure 22: Average I/O per Query on Connecticut Database for Different Algorithms

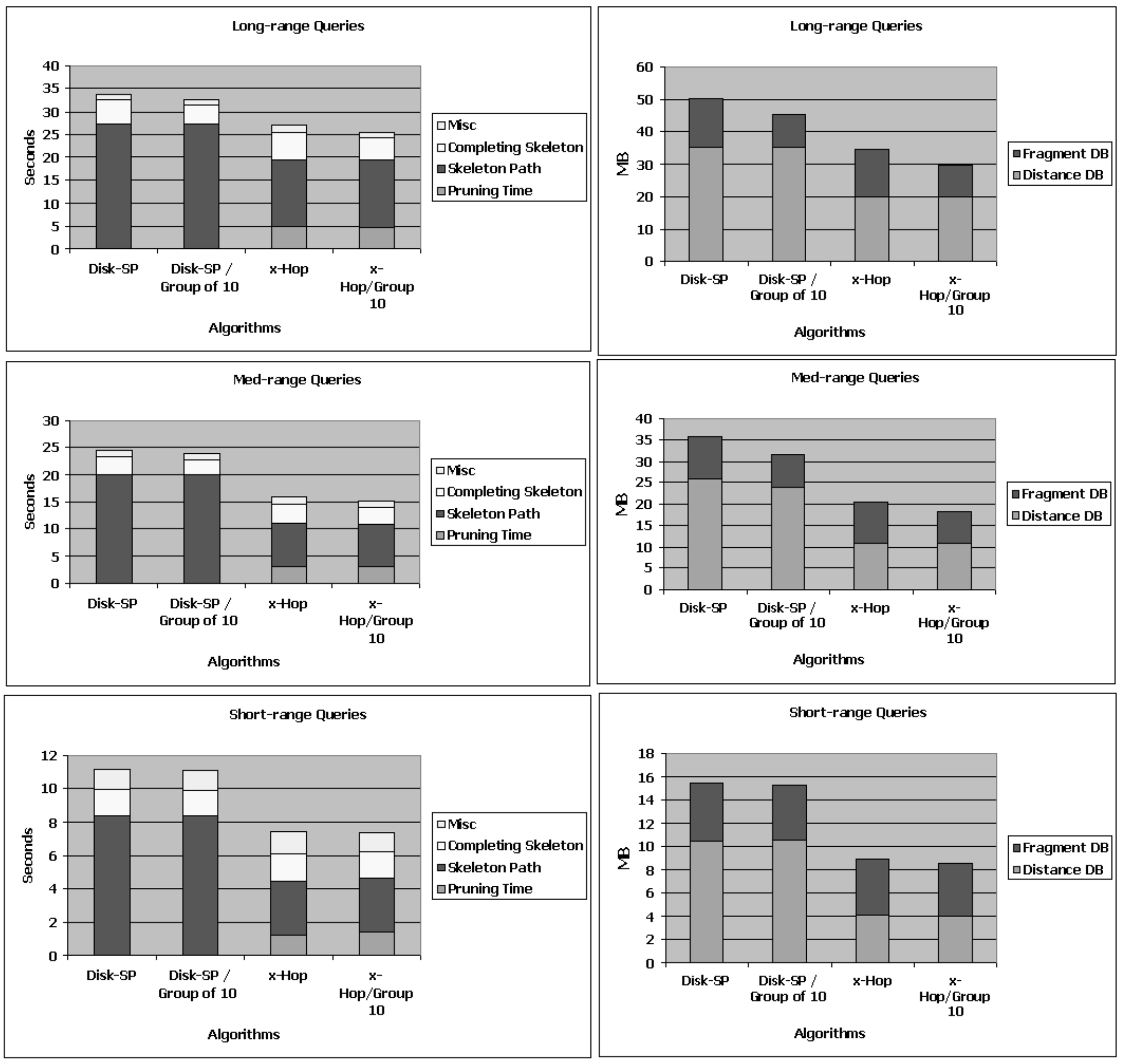


Figure 23: Average Calculation Time and I/O per Query on East5 Database for Different Algorithms