A Fast Unified Optimal Route Query Evaluation Algorithm*

Edward P.F. Chan & Jie Zhang
David R. Cheriton School of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
epfchan@uwaterloo.ca & janezhangj@hotmail.com

Abstract

We investigate the problem of how to evaluate, fast and efficiently, classes of optimal route queries on a massive graph in a unified framework. To evaluate a route query effectively, a large network is partitioned into a collection of fragments, and distances of some optimal routes in the network are pre-computed. Under such a setting, we find a unified algorithm that can evaluate classes of optimal route queries. The classes that can be processed efficiently are called *constraint preserving* (*CP*) which include, among others, shortest path, forbidden edges/nodes, α-autonomy, and some of hypothetical weight changes optimal route query classes. We prove the correctness of the unified algorithm. We then turn our attention to the optimization of the proposed algorithm. Several pruning and optimization techniques are derived that minimize the search time and I/O accesses. We show empirically that these techniques are effective. The proposed optimal route query evaluation algorithm, with all these techniques incorporated, is compared with a main-memory and a disk-based brute-force *CP* algorithms. We show experimentally that the proposed unified algorithm outperforms the brute-force algorithms, both in term of CPU time and I/O cost, by a wide margin.

Categories and Subject Descriptors: H.2.4 [Database Management]: Systems - Query processing. General Terms: Algorithms, Performance, Experimentation.
Additional Key Words and Phrases: Optimal Route Queries, Distance Materialization, Route Query Evaluation.

1 Introduction

Consider a web-based road information system, like Yahoo!Maps or a moving object database [13], in which commuters issue queries to find optimal routes from current locations to their destinations. For such a system to be valuable, it is imperative that the route queries are answered fast, preferably in real-time.

Some of the most frequently asked route queries, such as shortest path queries, as well as Travelling Salesman and Hamiltonian Path queries, are examples of optimal route queries. Among all route queries, *shortest path* (*SP*) queries are the most fundamental, and have been studied extensively in the literature. There are many mature SP algorithms. However, when a graph is huge, and the main memory is not large enough to load the whole graph, these algorithms do not work properly. Even if the main memory is large enough to accommodate the graph, the running time could be long, and the system resources may not be utilized effectively. It is well-known that the SP problem can be solved in polynomial time, while some optimal route problems are NP-complete. Although many papers, such as [6, 9, 4], deal with the optimal route problems, most of them only solve a specific case. Moreover, it is difficult to extend their solutions to a disk-based environment.

In this paper, we focus on the problem of how to evaluate classes of optimal route queries under a disk-based framework. In particular, we are interested in finding, with the help of distance materialization, a general and fast algorithm that can evaluate classes of optimal route queries. Such an algorithm is found, and is derived from an existing SP query evaluation algorithm [2, 3]. We call the classes of optimal route queries that can be evaluated by the proposed algorithm *constraint preserving* (*CP*). *CP* optimal route queries include real-life queries such as "find a route from a to b with the minimal cost," "find an optimal path from a to b that does not go through the center core of the city," "find the best trip from a to b that does not pass a set of towns and cities, and avoid Highways 9, 10 and 20,", "find an optimal route from a to b such that the distance of between successive nodes (which could denote intersections, hotels, gas stations) is not greater than c," and "find an optimal route from a to b, assuming that Highway 401 traveling time is increased by 15% while the traveling time on Highway 400 is decreased by 10%."

In this work, we first prove that the generalized algorithm correctly evaluates a *CP* optimal route query. To minimize the search space, algorithms have been proposed in [2] to prune the network when evaluating an SP query. We show that these pruning algorithms can be generalized to *CP* optimal route queries. Furthermore, improvements are made on these pruning algorithms that optimize their execution

*An Extended Abstract appears in the Proceedings of ACM 16th International Conference on Information and Knowledge Management, Lisbon, Nov. 2007.

times. After that, we turn our attention to finding techniques for route query evaluation. Several techniques are found that could speed up the execution time or reduce the I/O cost. We show empirically that the proposed modified pruning algorithms, and the proposed route query evaluation techniques contribute to the improvement of the evaluation process. To evaluate the performance of the proposed route query evaluation algorithm, two real-life data sets and three *CP* optimal route query classes are used in our experiments. We show experimentally that the proposed *CP* route query evaluation algorithm, with all techniques incorporated, outperforms significantly, both in term of the execution time and I/O cost, a main-memory and a disk-based brute-force *CP* query evaluation algorithms.

In Section 2, we define some basic notation. In Section 3, we survey related work. In Section 4, we define the *CP* optimal route query classes, and show how they can be evaluated efficiently with a query evaluation algorithm. We also introduce techniques that can speed up the evaluation process. In Section 5, experimental results are presented. Finally, a summary is given in Section 6.

2 Definition and Notation

2.1 Basic Notation

A network such as a road system is denoted as a directed graph $G = (V, E, w)$, where V and E are the sets of nodes (vertices) and directed edges, respectively, and $w{:}E \rightarrow \mathcal{R}^{\geq 0}$, that is, w is a length function from the set of edges to a set of non-negative real numbers. Let p be a path in a graph G. Then $l(p)$ is the sum of the lengths of edges in path p. Since we are interested only in paths *without any loop*, unless otherwise stated, a path is a *simple* path. If a path p starts from node s and ends at node t, then p is an *s-t path*.

2.2 Optimal Route Queries

An optimal route query returns an SP in a graph G that satisfies certain constraint. Let θ be a constraint imposed on paths in a graph G. If θ is null (Λ), then any path in G is satisfying wrt θ. The constraint θ could be dependent on G, or could be a general condition on paths in G. Examples of general path constraints include the following: the number of nodes/edges in a path, or the length of a path must be less than certain constant c. Instances of constraints that are dependent on a graph G include: a path cannot pass through a set of specific edges/nodes in G, or a path must visit certain edges/nodes in G in a specific order.

An *optimal route query*, denoted as $Q(G,\theta,s,d)$, where G is a graph, s and d are two distinct nodes in G, and θ is a constraint imposed on paths in G. The *answer* to an optimal route query $Q(G,\theta,s,d)$ is a satisfying *s-d* path in G, wrt θ, and no other satisfying *s-d* path in G with a shorter length. An empty path is returned if no path in G satisfies the constraint θ. The answer to an optimal route query is called an *optimal route*. If θ is Λ, then $Q(G,\theta,s,d)$ is an SP from s to d in G. Unless confusion arises, we shall use $Q(G,\theta,s,d)$ to denote an optimal route query as well as the answer to the query. The *length* of an optimal *u-v* route in G, denoted as $SD(G,\ \theta,\ u,\ v)$, is defined as $l(Q(G,\theta,u,v))$, if an optimal path exists, and ∞ otherwise. An *optimal route query class* $Q(G,\theta)$ is the set of optimal route queries $\{Q(G,\theta,s,d) \mid s \text{ and } d \text{ are distinct nodes in } G\}$. The following are examples of optimal route query classes:

- **Shortest Path**: θ: Λ.
- **Forbidden Nodes/Edges**: θ: Nodes(Edges) in a path cannot be in a specific set S of nodes (edges, respectively) in a graph G.
- **α-autonomy**: θ: An edge in a path cannot have a length greater than α.
- **k-stops**: θ: There are exactly k-1 edges in a path.
- **Travelling Salesman**: θ: A path passes each node in N exactly once, where N is a subset of nodes in a graph G.
- **n-consecutive Nodes**: θ: Any n consecutive nodes in a path must be of the different colors, where $n \geq 2$, assuming that nodes in G are colored.
- **Hypothetical Weight Changes**: θ is defined with respect to a graph G', where G' is obtained by applying a set of edge weight changes to G.

The α-autonomy and k-stops optimal route query classes are proposed and studied in [11], while a generalization of Travelling Salesman which is called *Trip Planning Query* (*TPQ*) is discussed in [5].

3 Related Work

We survey some existing related work in Section 3.1. Since this work is based on a disk-based SP algorithm, we briefly discuss it in Section 3.2.

3.1 Previous Work

A framework is presented in [10] to address the issue of how to evaluate various types of spatial queries in the presence of a road network. The query classes investigated include *k-NN*, range, distance-join, and closest-pairs queries. To efficiently access nodes and edges in a network, the adjacency lists of the nodes closed in space are placed in the same disk page. To support queries that explore the spatial properties of a road network, an R-tree is constructed for the minimal bounding rectangles of the polylines in the network. With this framework, algorithms are proposed to answer the above-mentioned query classes. However, this work does not focus on route queries nor their evaluation.

Given a large set of points, and assuming that the distance between a pair of points is their Euclidean distance, the authors in [11] formulate and solve the α-autonomy and k-stops shortest path queries in spatial databases. To solve these problems, algorithms with pruning are introduced which optimize the search space. However, it is not clear how their techniques can be extended effectively to network environment in which the distance is given as the path length.

A class of optimal route queries known as *Trip Planning Query (TPQ)* is introduced and investigated in [5]. Given a set of points P, where each point belongs to a specific category, a source and a destination, TPQ retrieves the best trip that passes through at least one point from each category. This problem is known to be NP-hard. In this work, the authors propose several fast approximation algorithms for TPQ. A number of approximation algorithms with various *approximation ratios* are proposed and investigated. For different instances of the problem, one can choose the algorithms with the best approximation ratio. Since all these are approximation algorithms, an optimal trip is, however, not guaranteed.

Work has been done on efficient evaluation of SP queries on massive graphs. Several methods have been proposed to solve the SP problem under the disk-based framework. See for instance [1, 7, 3, 8, 2]. To solve the scalability problem, all of them use the graph partitioning technique, and consist of a *pre-processing* and a *query evaluation* phases. A distinct feature of all these approaches is to use data materialization to speed up the query evaluation process. During the pre-processing, a graph is first divided into a set of "fragments" which are stored on a disk. Each fragment is small enough to be loaded into the main memory, and is a unit of transfer from the disk to the main memory. Because a graph is massive, some pre-computations are performed on these fragments, and the resulting information is materialized, commonly on some disk-based structures. Having pre-processed the network, the system is ready to evaluate SP queries. During the query evaluation phase, these methods make full use of the pre-computed data to reduce the execution time and I/O accesses.

3.2 An SP Query Evaluation Algorithm

Since our proposed optimal route query evaluation algorithm is derived from an algorithm called *DiskSP* in [2, 3], we briefly outline the framework and the algorithm here.

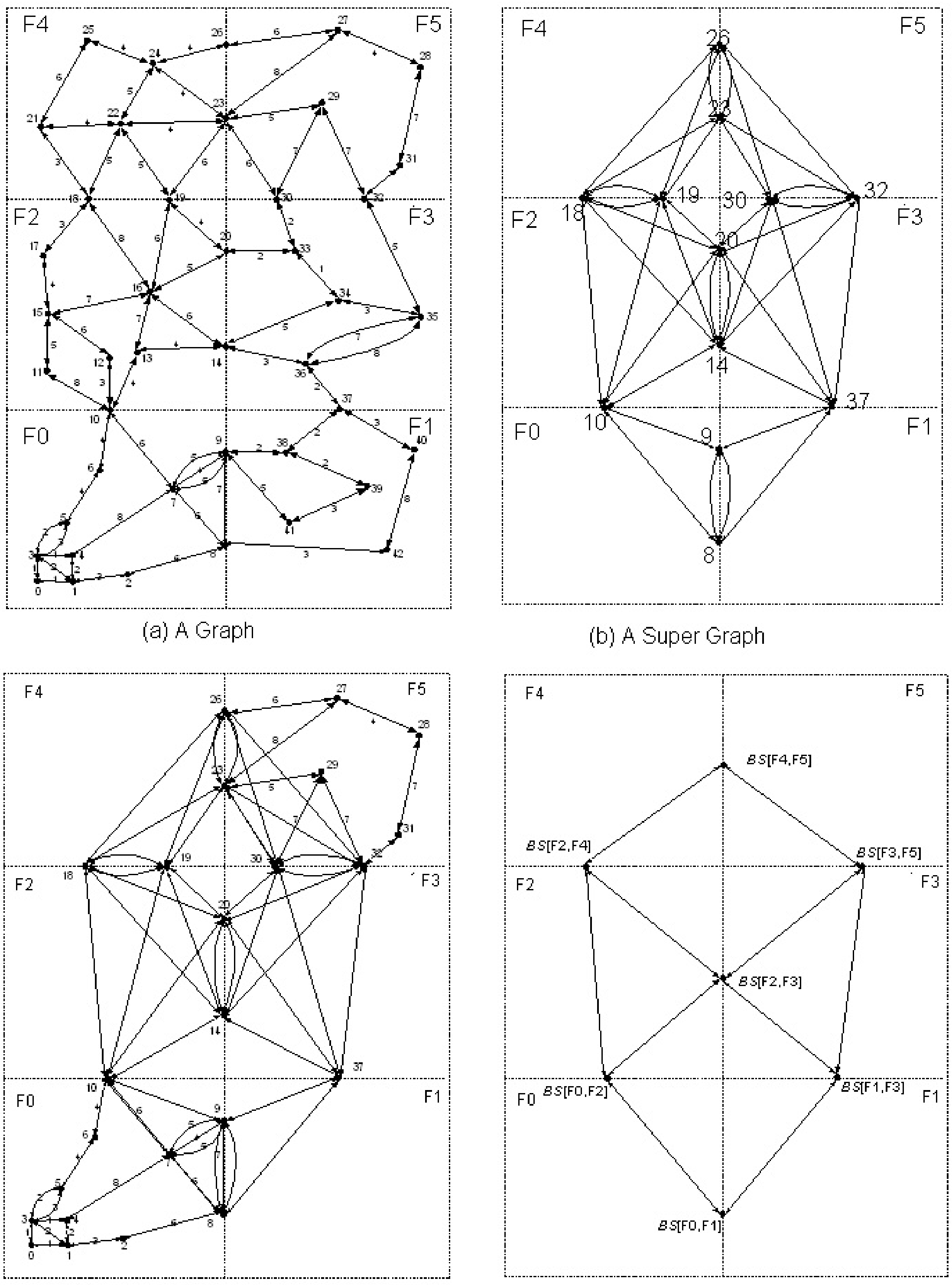
3.2.1 Graph Partitioning, Fragments, Super Graphs and Sketch Graphs

A graph G is assumed to be too large to be main-memory resident. It is first partitioned into a collection of fragments before queries are posted to the database. 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 G. A *partition* $P(G)$ 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 $\forall f\ \forall e{\in}E_f,\ w_f(e) = w(e)$. The resulting partition is stored in a disk-based structure, and is called a *fragment database*.

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 nodes in F. If two distinct fragments F_i and F_j are sharing at least one boundary node, then they are said to be *adjacent* to each other, and the set of boundary nodes shared by them is called a *boundary set*, denoted by $BS[F_i, F_j]$. Figure 1(a) shows a network is partitioned into six fragments with sever boundary sets. For instance, the subgraph containing nodes 0 to 10 forms a fragment F_0. Likewise, the subgraph for nodes 10 to 20 constitutes another fragment F_2. The boundary set $BS[F_0, F_1]$ consists of nodes 8 and 9.

Conceptually, once a graph is partitioned, one can apply a route query evaluation 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, this brute-force method may not be effective, especially if the search space is huge. For some classes of route queries, query evaluation can be speeded up by pre-computing some optimal distances. Given a route query class $Q(G,\theta)$, and for each fragment F in a partition, a *distance matrix* is created to record the distance of a local optimal path from one boundary node to the other. That is, for each pair of distinct boundary nodes u and v, the values $SD(F,\ \theta,\ u,\ v)$ and $SD(F,\ \theta,\ v,\ u)$ are recorded in a distance matrix. All these matrices collectively are called a *distance (matrix) database* for the query class $Q(G,\theta)$. The distance database for a query class $Q(G,\ \theta)$ of a graph partition $P(Q)$ is denoted as $DMDB(P(Q),Q(G,\ \theta))$, or $DMDB(Q(G,\ \theta))$, if the partition is understood from the context.

Given a partition $P(G)$ (of a graph G) and a query class $Q(G,\theta)$, boundary nodes and their pre-computed optimal distances in fragments give rise to a *super graph* for $Q(G,\theta)$. 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. Nodes and edges in a super graph are called *super nodes* and *super edges*, respectively. The length of a super edge $\langle u,\ v\rangle$ in a fragment F is $SD(F,\ \theta,\ u,\ v)$. The super graph for the partition in Figure 1(a) is shown in Figure 1(b). However, the lengths of super edges are not given explicitly. Super graphs capture the connectivity of boundary nodes, and can be used to find an SP between two boundary nodes in a partition.



(a) A Graph

(b) A Super Graph

(c) An Augmented Super Graph

(d) A Sketch Graph

Figure 1: An Example

Formally, a *super graph* for a query class $Q(G,\theta)$, denoted as $SG(P(G), Q(G,\theta)) = (V_s, E_s, w_s)$, of a graph partition $P(G) = \{F_1 = (V_1, E_1, W_1), F_2, \ldots, F_n\}$ is a directed graph and has the following properties: $V_s = \{v_b | \exists F_i, v_b$ is a boundary node in $F_i\}$, $E_s = \{\langle v_i, v_j \rangle_{F_k} | \exists F_k, v_i \neq v_j$ and they are boundary nodes in $F_k\}$, and $\forall\ e_{ij} \in E_s$, $w_s(e_{ij}) = w_s(\langle v_i, v_j \rangle_{F_k}) = SD(F_k, \theta, v_i, v_j)$. If the partition is understood from the context, from now on, a super graph for an optimal query class is denoted as $SG(Q(G,\theta))$.

Another useful concept is called sketch graphs. Sketch graphs are used to prune the search space. The connectivity of boundary sets in a partition can be captured with a sketch graph. Boundary nodes in a partition can be grouped into boundary sets. For instance, the set of boundary nodes shared by (distinct) fragments F_0 and F_1 in Figure 1(a), denoted as $BS[F_0, F_1]$, is the set of boundary nodes 8 and 9. The *sketch graph* consists of boundary sets as its nodes, and an edge from a node to another node if both nodes are in the same fragment. The edge lengths in a sketch graph are optionally specified. The sketch graph, with no edge lengths specified, for the partition in Figure 1(a) is shown in Figure 1(d). Formally, a *sketch graph* $K = (V_k,\ E_k,\ w_k)$ of a graph partition $P(G) = \{F_1, F_2, \ldots, F_n\}$ is a directed graph and has the following properties: $V_k = \{v_q \mid v_q$ corresponds to some boundary set $BS[F_i, F_j]\}$, $E_k = \{\langle v_i, v_j \rangle_{F_p} |\ v_i$ and v_j correspond to some distinct boundary sets in some fragment $F_p,\ p \in [1, n]\}$, and w_k is defined whenever is needed and is undefined otherwise.

Example 3.1 Suppose we have a road system, abstractly represented as a graph and is partitioned as shown in Figure 1(a). Edges denote street blocks and are bi-directional. 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. The shortest distances are materialized and stored in a *DMDB*($Q(G,\Lambda)$). To evaluate an SP query, an augmented super graph is constructed. An *augmented super graph* for an SP query $Q(G,\Lambda,s,d)$, denoted as $ASG(Q(G,\Lambda,s,d))$, is a super graph augmented with the source and destination fragments. Figure 1(c) is an augmented super graph for source node 0 and destination node 28.

Given an augmented super graph *ASG* for an SP query Q, we can search for an SP p for Q by searching on *ASG* with an algorithm like *Dijkstra's*. Once p is found, the actual SP from source to destination can be found by replacing each super edge in p with the corresponding SP in the fragment involved. A super graph can be implicitly represented by the source and destination fragments, and the *DMDB*($Q(G,\Lambda)$). Since path-finding algorithms, like *Dijkstra's*, process nodes locally, a matrix can be read into main memory whenever is needed. Clearly, each processed node still has some auxiliary data associated with it. This data must be stored in some disk-based data structure. Thus, the above method of finding an SP, which is called *DiskSP* in [2, 3], is scalable to a very large graph. █

In this method, there are two phases: *pre-processing* phase in which a graph is partitioned and distances of some SP's are computed, and *query processing* phase in which queries can be posted to the system, and answers are computed and returned to the user. We now describe the two phases in more detail, assuming that we are given an SP query class $Q(G,\Lambda)$.

3.2.2 Pre-Processing

With the help of an R-Tree, an arbitrary large graph G can be partitioned into a set of fragments. Fragments are stored on disks, and each is accessed as a unit during query processing. During the pre-processing phase and after the fragment database is created, the distance database *DMDB* and the sketch graph for the query class $Q(G,\Lambda)$ are constructed.

3.2.3 Query Processing

An SP query is evaluated by an algorithm called *DiskSP*. We describe conceptually how an SP query is evaluated here. Given an SP query $Q(G,\Lambda,s,d)$ and the distance database *DMDB*($Q(G,\Lambda)$), *DiskSP* evaluates a query in two steps: *skeleton path finding* and *skeleton edge filling.*

In the *skeleton path finding* phase, a disk-based variant of *Dijkstra's* algorithm is invoked to compute an SP p from s to d in an augmented super graph $ASG(Q(G,\Lambda,s,d))$. Conceptually, an augmented super graph is obtained by merging the source fragment S and the destination fragment D into the *super graph.* Because of the size of $ASG(Q(G,\Lambda,s,d))$, it is not physically constructed. Instead, the source and destination fragments are main-memory resident, and the corresponding boundary nodes and their distance matrices are read into main memory, whenever a part of the augmented super graph is processed. The *s-d* path p found in the first step is called a *skeleton path.* A skeleton path may contain edges from the super graph, and they are called *skeleton edges.* Each skeleton edge in fact represents an SP in some fragment for the two boundary nodes.

In the *skeleton edge filling* phase, for each skeleton edge of p, the actual SP in the corresponding fragment is computed, and then merge these paths into a complete path from s to d, which is the SP to query $Q(G,\Lambda,s,d)$.

3.3 Optimization and Search Space Pruning

Algorithm *DiskSP*, like *Dijkstra's*, suffers from the problem that many nodes are needlessly searched, in finding a skeleton path. This results in higher I/O cost and execution time. Pruning techniques are introduced to reduce the search space during the skeleton path finding phase. Two *sketch graph pruning* algorithms were introduced; one is based on the *boundary set distance matrix* (*BSDM*) and the other is based on *x-hop graphs* (*XHOP*) which results in two SP evaluation algorithms $DiskSP_{BSDM}$ and $DiskSP_{XHOP}$.

With these pruning techniques, a *BSDM* and x-hops graphs are pre-computed in the pre-processing phase.

With the introduction of pruning, an SP query evaluation algorithm consists of three phases: *sketch graph pruning*, *skeleton path finding*, and *skeleton edge filling*. In $DiskSP_{BSDM}$ and $DiskSP_{XHOP}$, the first phase is handled by a sketch graph pruning algorithm, while the last two phases are implemented by invoking *DiskSP*.

4 Our Contribution

In this work, we first show that *DiskSP* can be generalized to evaluate other optimal route queries. More specifically, we prove that certain classes of optimal route queries, which we call *constraint preserving* (CP), can be evaluated very fast with a unified route query evaluation algorithm. *CP* optimal route queries include many real-life optimal route queries. The *CP* optimal route query classes will be defined in Section 4.1. The generalized *DiskSP*, which we call *DiskCP*, will be discussed in Section 4.2. In Section 4.3, we show that two sketch graph pruning algorithms [2] mentioned in Section 3.3 can be improved and extended to prune sketch graphs for *CP* optimal route queries. In Section 4.4, we discuss several techniques that reduce the execution time and I/O cost for finding skeleton paths in *DiskCP*. The new algorithms, with improved sketch graph pruning and with all optimization techniques incorporated, are called $DiskCP_{BSDM}$ and $DiskCP_{XHOP}$.

4.1 CP Optimal Route Queries

It has been shown that SP query evaluation benefits significantly from distance materialization [2]. We now define a set of optimal route query classes that can be evaluated very fast with this method. To our best knowledge, the proposed algorithm is the first disk-based algorithm that can be used to evaluate classes of optimal route queries. Moreover, this set of optimal route query classes is the largest that can be evaluated very fast with distance materialization by a unified algorithm.

Consider the n-consecutive nodes query class. If a path p in G has fewer than n nodes, then p is not considered to be well-defined, and we said p is not in the *domain* of the constraints. A path p is *satisfying* wrt θ if p is in domain of θ and is evaluated to *true* wrt θ. θ is said to be *constraint preserving* (CP) wrt G, if (i) given any satisfying path p wrt θ in G, any sub-path q of p is also satisfying wrt θ, and (ii) given any sequence of connected paths p_1, p_2,..., p_n in G, where $n > 0$, such that each path p_i satisfies θ, then the path p=$p_1\diamond$ $p_2\diamond\ldots\diamond$ p_n also satisfies θ, where $\diamond$ is the path concatenation operator. Two consecutive paths are said to be *connected* if the end point of first is the start point of the second. A sequence of paths are *connected* if each consecutive pair is. The definition of *CP* captures the essence of when a query class can be evaluated fast under the graph partitioning framework with distance materialization. It remains to be seen if a fast algorithm can be found for classes of non-CP queries.

An optimal route query class $Q(G,\theta)$ is said to be *constraint preserving* (CP) if θ is *CP* wrt G. Clearly, *CP* optimal route query classes are a subset of optimal route query classes. Not every optimal route query class is *CP*. For instance, k-stops [11], trip-planning [5], and n-consecutive nodes query classes, where k and $n \geq 3$, are not *CP*, and thus cannot be evaluated with the proposed algorithm. On the other hand, SP, forbidden edges/nodes, α-autonomy [11], 2-consecutive nodes, and some of hypothetical weight changes query classes are *CP*. The reason that *CP* route query classes can be evaluated effectively is due to the following properties.

Corollary 4.1 *Let $Q(G,\theta)$ be a CP optimal query class, and p be a path in G. The path p is satisfying wrt θ iff every edge in p is satisfying wrt θ.*

Proof Follows from the definition of *CP* optimal route query classes. ∎

Let G_θ be a graph obtained from G by removing all edges not satisfying wrt θ. We call G_θ, G's *transformed graph* (wrt θ). Let $P(G)$ be G's partition. Let the partition obtained by removing all unsatisfying edges from $P(G)$, denoted as $P'(G_\theta)$, be the *transformed* partition (wrt θ) of $P(G)$. By construction, there is a 1-1 correspondence between fragments in $P(G)$ and $P'(G_\theta)$. Let F and F' be a corresponding pair of fragments from $P(G)$ and $P'(G_\theta)$, respectively. Then F' is obtained from F by removing all unsatisfying edges. To simplify the discussion, we assume from now on, whenever we talk about G and its transformed G', their partitions are $P(G)$ and $P'(G_\theta)$, respectively.

Corollary 4.2 *Let $Q(G,\theta)$ be a CP optimal query class, p be a path in G, and G_θ be G's transformed. The path p is satisfying wrt θ in G iff p is a path in G_θ.*

Proof Follows from Corollary 4.1 and from the definition of G's transformed. ∎

By Corollary 4.2, finding an optimal route in G is the same as finding an SP in the transformed graph G_θ. This implies the problem of finding optimal routes for *CP* query classes can be reduced to the problem of finding shortest paths with distance materialization.

Corollary 4.3 *Let G be a graph, $Q(G,\theta)$ be a CP optimal route query class, and G_θ be G's transformed. Then (i) $DMDB(Q(G,\theta))$ and $DMDB(Q(G_\theta,\Lambda))$ are identical, and (ii) $SG(Q(G,\theta))$ and $SG(Q(G_\theta,\Lambda))$ are the same.*

Proof By assumption on the partitions for G and G_θ, the set of boundary nodes and boundary sets in both partitions are identical. Consider any pair of boundary nodes s and t in a pair of corresponding fragments F and F'. By the construction of a transformed partition and by Corollary 4.2, an (local) optimal route s-t in F is a (local) SP in F'. Thus,

$SD(F,\ \theta,\ s,\ t)$ and $SD(F,\ \Lambda,\ s,\ t)$ are the same. Since the sets of boundary nodes in the corresponding fragments are identical, $DMDB(Q(G,\theta))$ and $DMDB(Q(G_\theta,\Lambda))$ are same. It follows that $SG(Q(G,\theta))$ and $SG(Q(G_\theta,\Lambda))$ are identical. ∎

4.2 DiskCP

As in *DiskSP*, *DiskCP* accepts seven parameters: s, d, S, D, F, M, K. Nodes s and d are source and destination, respectively, and S and D are their corresponding fragments. F and M are the fragment database and distance database for the query class $Q(G,\theta)$, respectively. K is a (pruned) sketch graph which is used to guide or limit the search of a skeleton path in evaluating a query. K is produced in the sketch graph pruning phase. *DiskCP* is a generalization of *DiskSP* since M is more general in *DiskCP*. In *DiskSP*, M is a distance database for the query class $Q(G,\Lambda)$, while the input M to *DiskCP* is a distance database for a *CP* optimal route query class $Q(G,\theta)$.

Given an SP query $Q(G,\Lambda,s,d)$, *DiskSP* can be considered as a *Dijkstra's* applied to the augmented super graph $ASG(Q(G,\Lambda,s,d))$. *DiskCP* is a generalization of *DiskSP*, and can be considered as a *Dijkstra's* applied to an augmented super graph $ASG(Q(G,\ \theta,\ s,\ d))$, where $Q(G,\theta)$ is a *CP* optimal route query class. However, there is one more change to *DiskSP* to obtain *DiskCP*: after a node in the source or destination fragment is closed, *DiskSP* relaxes all adjacent edges. However, in *DiskCP*, after a node in the source or destination fragment is closed, instead of relaxing all adjacent edges, only those *satisfying* adjacent edges (wrt θ) are relaxed. Adjacent edges considered include edges in the source and destination fragments, and super edges in the super graph. By construction, each super edge denotes an optimal path in some fragment, thus, each super edge is satisfying (wrt θ). *DiskCP* is a generalization of *DiskSP*, and is obtained with the above-mentioned modification from *DiskSP*.

We are now ready to show that *DiskCP* correctly evaluates *CP* optimal route queries. Conceptually, given a *CP* optimal route query $Q(G,\theta,s,d)$, *DiskCP* computes a skeleton path by applying *Dijkstra's*, starting from s, to an augmented super graph $ASG(Q(G,\ \theta,\ s,\ d))$. Let p be a skeleton path obtained from s to d in the augmented super graph $ASG(Q(G,\ \theta,\ s,\ d))$. We claim p is a skeleton path of the answer $Q(G,\theta,s,d)$.

Let G be a graph, and G' it's transformed. Consider the augmented super graphs for queries $Q(G,\theta,s,d)$ and $Q(G_\theta,\Lambda,s,d)$. By Corollary 4.3, the augmented super graphs $ASG(Q(G_\theta,\ \Lambda,\ s,\ d))$ and $ASG(Q(G,\ \theta,\ s,\ d))$ differ only on the unsatisfying edges (wrt θ) in the source and destination fragments. That is, if we remove all unsatisfying edges in the source and destination fragments from $ASG(Q(G,\ \theta,\ s,\ d))$, the two augmented super graphs are identical.

Consider now we apply *DiskSP* to $ASG(Q(G_\theta,\ \Lambda,\ s,\ d))$, and invoke *DiskCP* to $ASG(Q(G,\ \theta,\ s,\ d))$. Recall that *DiskCP* is obtained from *DiskSP* by relaxing only satisfying edges. This implies that the set of edges in source and destination fragments that can be relaxed by both algorithms are identical. Therefore, p is a skeleton path returned by *DiskSP* on $ASG(Q(G_\theta,\ \Lambda,\ s,\ d))$ precisely when p is a skeleton path returned by *DiskCP* on $ASG(Q(G,\ \theta,\ s,\ d))$.

Corollary 4.4 *Let $Q(G,\theta,s,d)$ be a CP optimal route query. DiskCP correctly evaluates Q.*

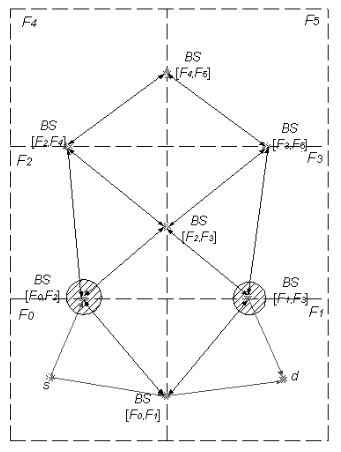
Proof Follows from the above argument, and from the correctness of *DiskSP*. ∎

4.3 Sketch Graph Pruning

It is necessary to prune a graph in order to minimize the search space. Two techniques were proposed in [2] to prune a sketch graph before a skeleton path is found. These techniques can be extended naturally to *CP* optimal route queries. Both techniques require some distance materialization: *Boundary Set Distance Matrix* ($BSDM$) and x-hop sketch ($XHOP$) graphs, respectively. These data are computed in the pre-processing phase. Given a pair of boundary sets in a sketch graph, $BSDM$ contains an upper and a lower bounds on the shortest distances between nodes in the pair. Likewise, an x-hop sketch graph can be used to derive an upper bound and a lower bound on the shortest distances between nodes in any pair of boundary sets. Given a *CP* query class, instead of finding SPs in the pre-computation, optimal routes are computed. With this modification, it can be proven that $BSDM$ and x-hop sketch graphs pruning can be generalized to any *CP* query class. During the sketch graph pruning, both algorithms in [2] prune boundary sets by searching the *whole* sketch graph. Instead of using an exhaustive search, we propose algorithms that based on the BFS which optimizes the pruning process. We illustrate the BFS on $BSDM$ with the following example.

Example 4.1 Figure 2 provides an example of the improved sketch graph pruning based on *BSDM*. Given a query Q and a boundary set BS, the table (which can be derived from the fragments and $BSDM$) gives the lower bounds from s to (any vertex in) BS and from (any vertex in) BS to d. Likewise, an upper bound $U(Q)$ from s to d can be computed and assumed to be 13.

First, check the adjacent boundary sets of s, $BS[F_0,\ F_2]$ and $BS[F_0,\ F_1]$ in source fragment F_0, where $L(Q,\ BS[F_0,\ F_2]) = L(s,\ BS[F_0,\ F_2]) + L(BS[F_0,\ F_2],\ d) = 17 > U(Q) = 13$ and $L(Q,\ BS[F_0,\ F_1]) = L(s,\ BS[F_0,\ F_1]) + L(BS[F_0,\ F_1],\ d) = 11 < U(Q) = 13$. Hence, prune $BS[F_0,\ F_2]$. Then, check the adjacent unvisited boundary set of $BS[F_0,\ F_1]$, which is $BS[F_1,\ F_3]$, and $L(Q,\ BS[F_1,\ F_3]) = L(s,\ BS[F_1,\ F_3]) + L(BS[F_1,\ F_3],\ d) = 15 > U(Q) = 13$. Thus, $BS[F_1,\ F_3]$ is pruned. After $BS[F_1,\ F_3]$ and $BS[F_0,\ F_2]$ are removed, no shortest path is available from s to other unvisited boundary sets. Consequently, we can prune them without computing any lower bounds. After pruning, $BS[F_0,\ F_1]$ is the only boundary set in the sketch graph. With the pruning approach in $DiskSP_{BSDM}$ [2], we need to compute 7 lower bounds, whereas with the new BFS method, and in this example, only 3 lower bounds are calculated. ▮



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

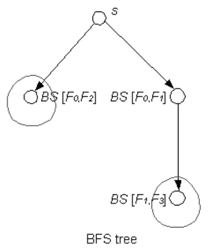


Figure 2: Sketch Graph Pruning With BSDM

Given a query $Q(G,\theta,s,d)$, the x-hop sketch graph pruning algorithm in [2] computes an upper bound on the length of $Q(G,\theta,s,d)$. After that, for each boundary set Y in the sketch graph, compute a lower bound $L(Q(G,\ \theta,\ s,\ d),\ Y)$, and see if the lower bound is greater than the upper bound. If it does, Y is pruned from the sketch graph. These lower bounds are derived from two shortest path trees (SPT's) which are in turn computed from a source-augmented x-hop α-sketch graph $SASG$ and a destination-augmented x-hop α-sketch graph $DASG$. The two SPT's contains all nodes in the sketch graph. The x-hop sketch graph pruning can be optimized, without exhaustively computing all lower bounds. Once an upper bound $U(Q(G,\ \theta,\ s,\ d))$ on the length of $Q(G,\theta,s,d)$ is known, the computation of the two SPT's for the lower bounds can be optimized by computing two partial SPT's as follows:

1. Given a SASG and starting from the source, compute a partial outgoing SPT until the *first* closed node whose distance is *just* greater than $U(Q(G,\ \theta,\ s,\ d))$. Let us mark the set of closed nodes in this partially computed SPT as *candidates.*

2. Given a $DASG$ and starting from the destination, computing an incoming SPT by relaxing and closing only nodes that are marked as *candidates* in $SASG$. For each closed node Y, compute the lower bound $L(Q(G,\ \theta,\ s,\ d),\ Y)$. If the lower bound is greater than the upper bound, mark Y as *non-candidate* in $DASG$, else mark it as *candidate.* It can be proven that only candidate boundary sets in $DASG$ need to be searched for an optimal query.

4.4 Techniques for Fast Skeleton Path Finding

Another contribution of this work are several techniques for a faster computation of a skeleton path.

4.4.1 Query Super Graph

Under this framework, *sketch graph pruning* is the first phase in finding an optimal path. During the pruning phase, an outgoing (local) SPT SPT_s rooted at source s in source fragment S, and an incoming (local) SPT_d rooted at destination d in destination fragment D are first computed. Hence, the optimal distances from s to any vertex in S, and the optimal distances from any vertex in D to d have been calculated. These computed optimal distances could be used to facilitate the second phase, the *skeleton path finding* phase.

Given an optimal query $Q(G,\theta,s,d)$, we can build a *query super graph* for the query from the super graph as the following:

1. Add the new nodes s and d into the super graph, if they are not already there.

2. For each boundary vertex v in S, insert a super edge from s to v, if it is not already there, into the super graph with distance $SD(S,\ \theta,s,v)$.

3. For each boundary vertex u in D, insert a super edge from u to d, if it is not already there, into the super graph with distance $SD(D,\ \theta,\ u,d)$.

4. When s and d are in the same fragment, insert a super edge from s to d, if it is not already there, into the super graph with distance $SD(S,\ \theta,\ s,\ d)$.

The differences between an augmented super graph and a query super graph lie in the source fragment and the destination fragment. The former includes all nodes and edges in S and D, while the latter contains only the outgoing super edges of s inside S, and only the incoming super edges of d inside D. Computing a skeleton path in a query super graph is faster than that in its corresponding augmented super graph.

4.4.2 Successor Fragment Relaxation

Given a *CP* query $Q(G,\theta,s,d)$, conceptually, *DiskCP* applies *Dijkstra's* algorithm on the (pruned) query super graph to compute a skeleton path from s to d. It iteratively selects a vertex u, with the minimum optimal distance from s, among all open vertices in the (pruned) super graph, adds it into a closed vertex set, then retrieves all adjacent edges of u and does relaxation on them. If u is a boundary vertex, then all adjacent super edges of u are retrieved as well. Let $BS[F_1, F_2]$ be the boundary set holding u, which is selected and closed in some i^{th} iteration. Obviously, all boundary vertices in fragments F_1 and F_2 (except u) are $u's$ adjacent vertices in super graph and are relaxed in the i^{th} iteration.

Let a skeleton path p_s be $\langle e_1, e_2, \ldots, e_m\rangle$, where $m \geq 2$. Let $e_{i-1} = \langle v_x, v_y\rangle_{F_k}$ and $e_i = \langle v_y, v_z\rangle_{F_{k'}}$, where e_{i-1} and e_i are skeleton edges, for some $i \in [2, m]$. The vertex sequence $\langle v_x, v_y, v_z\rangle$ is a sequence of boundary vertices. Then, the *predecessor* of boundary vertex v_y in path p_s is v_x and the *successor* of v_y is v_z. Likewise, the adjacent fragments F_k and $F_{k'}$ are said to be *predecessor fragment* and the *successor fragment* of v_y, respectively.

Given a *CP* query $Q(G,\theta,s,d)$, there is a nice triangle inequality property for any boundary vertices x, y, z in a fragment F: $SD(F,\theta,x,y) + SD(F,\theta,y,z) \geq SD(F,\theta,x,z)$. This property follows from the fact that, given a *CP* query class, the existence of optimal paths *x-y* and *y-z* implies the existence of a satisfying path *x-z*. Let u be a boundary vertex in a fragment F. Assume that p is a skeleton path from s to u. Let F be the predecessor fragment of u on a skeleton path p. With the triangle inequality property, we can prove that it is unnecessary to relax the outgoing super edges of u inside its *predecessor* fragment F. The way to prove this claim is to show that, even if those edges were relaxed, the optimal distance of the head of a relaxed edge will not be improved. We denote *x.distance* as the potential optimal distance from source s to a vertex x in a given graph G. Once x is closed, $x.distance = SD(G, \theta, s, x)$.

Lemma 4.5 *Given a CP optimal route query $Q(G,\theta,s,d)$, suppose we apply Dijkstra's algorithm, starting from source s, to an augmented super graph or to a query super graph. Suppose in the i^{th} iteration of Dijkstra's algorithm, a boundary vertex u is selected to be closed, where $i \in [2, n]$, and n is the number of iterations in the algorithm. Denote p as the skeleton path computed so far from source s to u in the graph. Let $e_{i-1} = \langle v, u\rangle_{F_l}$ be the last edge in p, and e_{i-1} is a skeleton edge. Then it is unnecessary to relax any super edge of u inside F_l in the i^{th} iteration.*

Proof Assume the boundary vertex v is closed in the j^{th} iteration. Obviously the fragment F_l is the successor fragment of v on p and we have $1 \leq j < i$. When v is closed, we do relaxation on all outgoing super edges of v, including those inside F_l, in the j^{th} iteration. Hence, during the relaxation process of v in the j^{th} iteration, for any boundary vertex x in F_l, if $SD(G,\theta,s,v) + SD(F_l,\theta,v,x) < x.distance$, then $x.distance$ is updated to $SD(G,\theta,s,v) + SD(F_l,\theta,v,x)$. As a result, at the end of the j^{th} iteration, $x.distance \leq SD(G,\theta,s,v) + SD(F_l,\theta,v,x)$. Since $j < i$, in the i^{th} iteration, we still have $x.distance \leq SD(G,\theta,s,v) + SD(F_l,\theta,v,x)$. According to the triangle inequality property, $SD(F_l,\theta,v,x) \leq SD(F_l,\theta,v,u) + SD(F_l,\theta,u,x)$. Since $SD(G,\theta,s,u) = SD(G,\theta,s,v) + SD(F_l,\theta,v,u)$, we have $x.distance \leq SD(G,\theta,s,v) + SD(F_l,\theta,v,x) = SD(G,\theta,s,u) - SD(F_l,\theta,v,u) + SD(F_l,\theta,v,x) \leq SD(G,\theta,s,u) + SD(F_l,\theta,u,x)$. Consequently, u is not the predecessor of a boundary vertex x in its predecessor fragment F_l on any skeleton path from s to x. Thus, it is unnecessary to relax any outgoing super edge of u inside F_l. ∎

When a node u is closed and its adjacent nodes are relaxed, the adjacent super edges are retrieved by reading in matrices in the *DMDB*. Computation is done on *u's* adjacent nodes to update their potential distances. With the concept of successor fragments, some of these adjacent super nodes and super edges are not retrieved. As a result, computation is reduced. The I/O accesses of the *DMDB* could potentially be reduced. However, the cache size has much influence on the improvement since the distance matrices involved could be in the cache if it is of a reasonable size.

4.4.3 Dynamic Boundary Vertex Pruning

Algorithms in [2] select a boundary vertex or a vertex inside the source fragment or the destination fragment to be closed, and then relax its outgoing edges iteratively. However, we observe that when a boundary vertex $v \in Y$ is selected, where Y is the boundary set holding v, and if v.distance+$L(Y,d,Q(G,\theta)) > U(Q(G, \theta, s, d))$, where $L(Y,d,Q(G,\theta))$ and $U(Q(G, \theta, s, d))$ are the lower bound from Y to the destination d and an upper bound on *s-d* path, respectively, then any optimal route from s to d will never go through v, and thus v is eligible to be pruned. It is worth noting that the lower bounds and upper bounds have been computed during the sketch graph pruning phase. In addition, as we will prove later, all currently non-closed vertices of Y can be also pruned. Since this technique reduces the number of boundary nodes that need to be closed, it has the advantages of reducing both the skeleton path computation time and *DMDB* accesses.

Lemma 4.6 provides the correctness proof of the dynamic boundary vertex pruning.

Lemma 4.6 *Let vertex u be the closed boundary vertex in the i^{th} iteration of the DiskCP algorithm, where $i \in [2, n]$ and n is the number of iterations in the algorithm. Let Y be u's host boundary set. If u.distance+$L(Y, d, Q(G,\theta))$ $> U(Q(G, \theta, s, d))$, then u and all the currently non-closed vertices in Y can be pruned.*

Proof After u is closed in the i^{th} iteration, u.distance is $SD(G, \theta, s, u)$. In addition, $SD(G, \theta, u, d) \geq L(Y,d,Q(G,\theta))$. Hence, we have $SD(G, \theta, s, u) + SD(G, \theta, u, d) \geq u$.distance + $L(Y,d,Q(G,\theta)) > U(Q(G, \theta, s, d)) \geq SD(G, \theta, s, d)$. Consequently, the length of any path from s to d via u is greater than $SD(G, \theta, s, d)$; that is, u is eligible for pruning. Let v be a non-closed boundary vertex of Y in the i^{th} iteration of *DiskCP*. $SD(G, \theta, s, v)$ must be greater than or equal to u.distance, because u is closed before v is. Thus, $SD(G, \theta, s, v) + SD(G, \theta, v, d) \geq u$.distance+$L(Y,d,Q(G,\theta)) > U(Q(G, \theta, s, d)) \geq SD(G, \theta, s, d)$. Therefore, v can also be pruned. ∎

5 Experiments

Section 4.3 presents improved versions of sketch graph pruning, and the resulting query evaluation algorithms are called

DiskCP$_{BSDM}$ and *DiskCP*$_{XHOP}$, respectively. Section 4.4 describes two optimization techniques to improve the running time and I/O cost of finding a skeleton path. Let us call the algorithm with all these sketch graph pruning and optimization techniques incorporated *DiskCP*. That is, *DiskCP* could either be *DiskCP*$_{BSDM}$ or *DiskCP*$_{XHOP}$, depending on the sketch graph pruning algorithm used, and with all optimization techniques incorporated.

Since there is no comparable algorithm, *DiskCP* is compared with a main-memory and a disk-based *CP* algorithms without any data materialization. We shall call them the *main-memory brute-force* (*MMCP*$_{BF}$) and the *disk-based brute-force* (*DiskCP*$_{BF}$) *CP* route query evaluation algorithms. We briefly outline these brute-force algorithms in Section 5.4. We first describe the environment of the experiments in Section 5.1, and then present the experimental results in Sections 5.2 to 5.4.

5.1 Environment, Data, and Query Sets

The PC for testing is a Pentium IV 1.6 GHz system with 1GB DDR, and the hard drive is an Ultra ATA/100 at 7,200 rpm. The operating system is Microsoft Windows Server 2000 with SP4. All algorithms are implemented with Java 1.4.1.

The road system of Connecticut, denoted as *CT*, from Tiger/Line file [12] is chosen as our first test case. It consists of around 190,000 edges and 160,000 nodes. CT is small enough to be loaded into the main memory and yet large enough to test the proposed main-memory and disk-based algorithms. To test 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 case. East5 is composed of the road systems of Connecticut, Massachusetts, New Jersey, New York, and Pennsylvania. It consists of more than three million edges and two and half million nodes. To make a homogeneous environment, we set the Java Virtual Machine (JVM), for CT and East5 test cases, to 512MB and 1GB, respectively. It has been shown that, for *DiskSP*, the optimal fragment size for CT and East5 are 1000 nodes and 2500 nodes, respectively [2]. On average, each such a CT fragment occupies 145KB while an East5 fragment is about 272KB. These fragment size databases are used in the experiments. The size of the fragment databases for CT and for East5 are 20MB and 315MB, respectively, while the size of distance databases *DMDB* for CT and for East5 are 2 MB and 40 MB, respectively. Because the sketch graph pruning algorithms are designed for small and large graphs, for CT, the algorithm *DiskCP*$_{BSDM}$ is called while for East5, *DiskCP*$_{XHOP}$ is invoked.

For *DiskCP*, the most I/O intensive structure is *DMDB* and its cache size is set to 25%. Since at most two fragments are needed during query evaluation, the cache size of a fragment database is set so that at most two fragments, at any time, are main-memory resident. All other data structures consume a relatively small amount of memory, and their cache sizes are set to 100% [2].

Queries with different ranges are investigated, and they are: *long, medium*, and *short.* The long-range queries are more than 66% of the longest possible shortest distance in the graph, the short-range ones are less than 33%, while the medium-range ones are between 33% and 66%. Each query set is randomly generated, and the number in a set will be given later in each experiment. Each testing result in this work is the average of all queries in a query set; each query is executed once.

5.2 Sketch Graph Pruning Evaluation

In Section 4.3, two improved algorithms on sketch graph pruning are proposed. In this section, we evaluate their performances by comparing with the exhaustive approaches proposed in [2]. All *CP* route query classes should benefit from sketch graph pruning. As explained in Section 4.2, all *CP* query classes can be reduced to the SP query class. Thus, SP query class is used in this experiment. The result obtained for SP query class should carry over to other *CP* route query classes. Since all pruning algorithms, given an optimal route query and the same pre-computed data, produce the same pruned sketch graph, we evaluate the pruning algorithms on their execution times. It is worth noting that the graph involved in sketch graph pruning is main-memory resident.

All these pruning algorithms, given an optimal route query, require the computation of an upper bound and numerous lower bounds. Since the cost of computing the upper bound is the same for all algorithms, they differ on the cost of lower bound computations. We shall denote each such a computation a *lower bound computation.* The execution time of a pruning algorithm is proportional to the number of lower bound computations, and thus it is used in evaluating different pruning algorithms. Each query set consists of 100 randomly generated queries. Since the original sketch graph pruning algorithms as proposed in [2] employ an exhaustive search, while our proposed pruning algorithms are based on the BFS, we shall denote them, in this section, as *exhaustive* and *BFS*, respectively. Table 1 shows the performances of different pruning algorithms in term of the number of lower bound computations. The experimental result in Table 1 shows that, with our proposed BFS pruning algorithms, the number of lower bound computations are reduced drastically. The reduction varies from 30% for long-range queries to 80% for short-range queries.

BSDM	Short	Med.	Long	XHOP	Short	Med.	Long
Exh.	694	694	694	Exh.	6132	6132	6132
BFS	114	219	399	BFS	1174	3027	4364

Table 1: Average Number of Lower Bound Calculations Per Skeleton Path Computation With a BFS and With an Exhaustive Search

5.3 Skeleton Path Finding Techniques Evaluation

In Section 4.4, we propose three techniques to improve the performance during the skeleton path finding phase, and they are: (i) We propose to conduct the skeleton path computation on a *query super graph* (QSG); rather than on an *augmented super graph* (ASG). (ii) Instead of relaxing all outgoing super edges of a closed boundary vertex, we propose to relax only the outgoing super edges of a closed boundary vertex in its *successor fragment* (SF). (iii) We introduce a *dynamic pruning* (DP) of boundary vertices.

In order to determine the effect individually, we investigate them separately. Since the proposed techniques are for improvement on the *skeleton path finding*, the discussion in this subsection will concentrate on this particular aspect only. For execution time, the dominant costs are graph-related and queue operations. For I/O cost, the most significant is the accesses to *DMDB*. Each query set consists of 100 randomly generated queries.

CT	Short	Med.	Long	East5	Short	Med.	Long
ASG	15236	25382	34303	ASG	113972	264979	368811
QSG	7628	15977	23589	QSG	88786	223174	315415

Table 2: Average Number Queue Operations Per Skeleton Path Computation With Augmented Super Graph and With Query Super Graph

The advantages of using a query super graph, instead of an augmented super graph, in computing a skeleton path are that there is no 'merging' of source and destination fragments with the super graph, and the size of the graph involved is likely smaller. As a result, the algorithm is less complex, and the computation time is shorter. Table 2 shows the number of queue operations performed by both algorithms. For CT (East5) data set, the reduction ranges from 30% (15%) for long to 50% (22%) for short queries. However, this technique has no influence on *DMDB* accesses.

The successive fragment relaxation improves the running time of a skeleton path computation by reducing the number of boundary edges that it relaxes. When a closed node is relaxed, adjacent nodes are accessed, computation and queue operations are performed on them. Table 3 illustrates that, in both CT and East5 data sets, the successor fragment relaxation technique reduces the number of times boundary nodes are accessed (which is the same as the number of times adjacent edges are relaxed) by more than 40%. Consequently, the skeleton path computation time is reduced as well. Since the distance matrix for adjacent boundary nodes are likely in cache when a boundary node is closed, this technique does not improve much on the *DMDB* accesses.

Dynamic boundary node pruning results in a fewer number of boundary nodes need to be processed. Table 4 shows the reduction in the number of closed boundary nodes. It is

CT	Short	Med.	Long	East5	Short	Med.	Long
No SF	0.07M	0.15M	0.22M	No SF	1.2M	3M	4.2M
With SF	0.04M	0.09M	0.13M	With SF	0.7M	1.7M	2.5M

Table 3: Average Number (in Millions) of Times Boundary Nodes are Accessed Per Skeleton Path Computation With and Without SF Relaxation

worth noting that the number of nodes closed is the number of iterations in the skeleton path finding algorithm. For both the CT and East5 data sets, the reduction is about 30%, and is independent of the query type.

CT	Short	Med.	Long	East5	Short	Med.	Long
No DP	320	791	1690	No DP	4101	11844	22899
With DP	226	530	1190	With DP	2966	8741	16546

Table 4: Average Number of Closed Boundary Vertices Per Skeleton Path Computation With and Without DP

CT	Short	Med.	Long	East5	Short	Med.	Long
No DP	18	36	69	No DP	108	311	592
With DP	16	30	57	With DP	84	242	452

Table 5: Average Number of Distance Matrix I/O Calls Per Skeleton Path Computation With and Without DP

Fewer number of closed boundary nodes implies shorter computation time and fewer *DMDB* accesses. The improvement of the I/O calls to *DMDB* is illustrated in Table 5. It shows the reduction of distance matrix I/O calls for CT (East5) data set ranges from 11% (22%) for short queries to 17% (24%) for long queries. Thus, this technique reduces the I/O cost of skeleton path computation, and its benefit increases proportionally with the size of the graph and the length of a query.

5.4 Evaluation With Distance Materialization

In this section, we shall show that *DiskCP* outperforms, by a wide margin, both the main memory and disk-based brute-force algorithms $MMCP_{BF}$ and $DiskCP_{BF}$. This shows the desirability of *DiskCP*. This also demonstrates that distance materialization is essential in fast *CP* route query evaluation. Since East5 is too large to be main-memory resident, only *DiskCP* and $DiskCP_{BF}$ are compared for this data set.

$MMCP_{BF}$ is obtained from main-memory *Dijkstra's* with a slight modification. Given a *CP* query $Q(G,\theta,s,d)$, and after a node is closed during an iteration, instead of relaxing all adjacent edges, $MMCP_{BF}$ only relaxes all *satisfying* adjacent edges wrt θ. So in effect, $MMCP_{BF}$ is *Dijkstra's* applies to $G's$ transformed G_θ. By Corollary 4.2, $MMCP_{BF}$ correctly computes the answer to a *CP* query.

Once a graph is partitioned into a collection of fragments, one can apply a *Dijkstra*-like algorithm onto fragments to compute a *CP* optimal route. In such an algorithm, no precomputed data such as distance materialization is used to speed up the search process. In $DiskCP_{BF}$, the search for a path starts from the source by reading in the source fragment and its auxiliary data structure. Nodes are extracted from the priority queue iteratively. In each iteration, the extracted node is closed, and all its adjacent satisfying edges with respect to θ are relaxed.

It takes a relatively long time for brute-force algorithms to execute a *CP* query. Since East5 is significantly larger than CT, the number of randomly selected queries for a query set for CT and for East5 are 30 and 15, respectively.

Three optimal route query classes are chosen in our evaluation: *SP*, *forbidden edges*, and *forbidden nodes* queries. These three sets are chosen because of their importance and/or potential real-life applications. For *forbidden edges* queries, 1% and 0.1% of edges in CT and East5, respectively, are randomly selected to form the forbidden edges set. No optimal route can pass through any edges in the forbidden set. For *forbidden nodes* queries, five nodes or seeds are first selected; one in the center of the graph and the remaining four locate at the center of the four quadrants of the graph. Each of the five seeds selected is used to generate a cluster of forbidden nodes. Starting with a seed, a BFS is performed to locate all forbidden nodes in a cluster. The search is terminated when the number of nodes in a cluster reaches $(x \times V)/5$, where V is the number of nodes in the graph. The values of x are 0.2% and 0.01% for CT and East5, respectively. For forbidden nodes queries, no optimal route is allowed to pass through a node in any of the five clusters.

For $DiskCP_{BF}$, we measure the amount of I/O performed on fragment databases and on the auxiliary data required during its execution. We observe that whenever a fragment is accessed, its auxiliary data are also retrieved. Consequently, for $DiskCP_{BF}$, the cache sizes for fragment database and for auxiliary database are set to the same value.

We experiment several cache sizes for CT and for East5 data sets, and we found the optimal cache sizes for $DiskCP_{BF}$ with CT and East5 are 50% and 20%, respectively. These cache sizes are used in the rest of the experiments. The cache sizes for disk-based structures in *DiskCP* are stated in Section 5.1. Due to the nature of these algorithms, the memory requirement by $DiskCP_{BF}$ are significantly larger than that by *DiskCP*.

Queues are an important part for all these algorithms. Thus, the number of queue operations performed is a good indication of the complexity of an algorithm. Consequently, in addition to execution time, the number of queue operations are also compared among these algorithms.

5.4.1 SP Query Class

Table 6 compares the I/O of the two disk-based *CP* algorithms: *DiskCP* (CP) and $DiskCP_{BF}$(BF). The amount of I/O by *DiskCP* for CT (East5) data set, relative to $DiskCP_{BF}$, are reduced by 91% (95%) for short queries and 95% (97%) for long queries. Thus, *DiskCP* requires only a small fraction of I/O accesses in evaluating an SP query when compared with $DiskCP_{BF}$.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	6.5	14.3	24.85	BF	92	321	470
CP	0.567	0.818	1.283	CP	4.4	9.8	16

Table 6: Average I/O (MB) Per SP Query

Table 7 records the average number of queue operations by all three algorithms. Relative to $MMCP_{BF}$(MM), the average numbers of queue operations of *DiskCP* on short, medium, and long queries on CT data set are reduced by 85%, 89%, and 88%, respectively. With respect to $DiskCP_{BF}$, the reduction of queue operations on short, medium and long queries on CT (East5) are 91% (95%), 92% (97%) and 92% (96%), respectively.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	0.12M	0.24M	0.4M	BF	1.5M	5M	6.8M
CP	0.01M	0.02M	0.03M	CP	0.07M	0.16M	0.26M
MM	0.07M	0.19M	0.29M				

Table 7: Average Number (in Millions) of Queue Operations Per SP Query

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	6.01	12.28	20.11	BF	103	607	820
CP	0.49	0.86	1.52	CP	4.02	9.1	15
MM	2.74	4.37	6.00				

Table 8: Average Query Evaluation Time (Seconds) Per SP Query

Table 8 shows the running time for all three algorithms. The execution time confirms the results obtained in Tables 6 and 7. Over all query sets, $DiskCP_{BF}$ has the worst running time while *DiskCP* performs the best. For CT data set, the query evaluation time of *DiskCP*, compared to that required by $MMCP_{BF}$, is about 18% for short and up to 25% for long queries. Compared to $DiskCP_{BF}$, *DiskCP* requires about 8% of $DiskCP_{BF}$ execution time to evaluate a query, independent of the query sets. For East5 data set, the reduction of execution time is even more dramatic. For short, medium and long queries, *DiskCP* requires only 4%, 1.5% and 2%, respectively, of time needed by $DiskCP_{BF}$. Thus, with a relatively small amount of distance materialization and with the proposed optimization, the *CP* query evaluation time and I/O can be reduced drastically. This result also demonstrates the scalability of *DiskCP*.

5.4.2 Forbidden Edges Query Class

Table 9 shows the amount of I/O performed by the two disk-based *CP* algorithms. The amount of I/O by *DiskCP* for CT (East5) data set, relative to $DiskCP_{BF}$, are reduced by 81% (91%) for short queries and 86% (93%) for long queries.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	6.48	14.26	24.9	BF	92.25	321.4	449
CP	1.202	2.02	3.49	CP	8.04	19.45	33.33

Table 9: Average I/O (MB) Per Forbidden Edge Query

Table 10 records the average number of queue operations by all algorithms. Relative to $MMCP_{BF}$, the average numbers of queue operations of *DiskCP* on short, medium, and long queries on CT data set are reduced by 85%, 89%, and 89%, respectively. With respect to $DiskCP_{BF}$, the reduction of queue operations on short, medium and long queries on CT (East5) are 90% (96%), 93% (97%) and 92% (96%), respectively.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	0.12M	0.24M	0.4M	BF	1.5M	5M	6.8M
CP	0.01M	0.018M	0.03M	CP	0.06M	0.16M	0.27M
MM	0.08M	0.17M	0.28M				

Table 10: Average Number (in Millions) of Queue Operations Per Forbidden Edge Query

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	6.1	12.4	20.9	BF	111	577	942
CP	0.75	1.05	1.81	CP	4.65	10.6	18.3
MM	2.79	4.15	5.81				

Table 11: Average Query Evaluation Time (Seconds) Per Forbidden Edge Query

Table 11 shows the running time for all three algorithms. For CT data set, the query evaluation time of *DiskCP*, compared to $MMCP_{BF}$, is about 27% for short and up to 31% for long queries. Compared to $DiskCP_{BF}$, *DiskCP* requires only 12% for short and 8% for medium and long queries of $DiskCP_{BF}$ execution time to evaluate a query. For East5 data set, the reduction of execution time is even more significant. For short, medium and long queries, *DiskCP* requires only 4%, 2% and 2%, respectively, of time needed by $DiskCP_{BF}$.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	7.18	14.34	24.9	BF	92.42	321.28	448.61
CP	1.21	1.95	3.34	CP	8.15	25.08	44.52

Table 12: Average I/O (MB) Per Forbidden Node Query

5.4.3 Forbidden Nodes Query Class

From Table 12, the amount of I/O accessed by *DiskCP* for CT (East5) data set, relative to $DiskCP_{BF}$, are reduced by 83% (91%) for short queries and 87% (90%) for long queries.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	0.13M	0.24M	0.4M	BF	1.5M	5M	6.8M
CP	0.01M	0.02M	0.03M	CP	0.06M	0.2M	0.36M
MM	0.08M	0.17M	0.28M				

Table 13: Average Number (in Millions) of Queue Operations Per Forbidden Node Query

Table 13 records the average number of queue operations by all three algorithms. Relative to $MMCP_{BF}$, the average numbers of queue operations of *DiskCP* on short, medium, and long queries on CT data set are reduced by 86%, 90%, and 89%, respectively. With respect to $DiskCP_{BF}$, the reduction of queue operations on short, medium and long queries on CT (East5) are 91% (96%), 93% (96%) and 93% (95%), respectively.

CT	Short	Med.	Long	East5	Short	Med.	Long
BF	6.67	12.4	20.9	BF	106	574	866
CP	0.577	0.852	1.57	CP	4.3	12.2	21.4
MM	2.79	4.12	5.8				

Table 14: Average Query Evaluation Time (Seconds) Per Forbidden Node Query

Table 14 shows the running time of different ranges of queries for all three algorithms. For CT data set, the query evaluation time of *DiskCP*, relative to $MMCP_{BF}$, is about 21% for short and up to 27% for long queries. Compared to $DiskCP_{BF}$, *DiskCP* requires only 9% for short and 8% for long queries of $DiskCP_{BF}$ query evaluation time. For East5 data set, and for short, medium and long queries, *DiskCP* requires only 4%, 2% and 3%, respectively, of time needed by $DiskCP_{BF}$.

6 Conclusion

We have studied the problem of how to evaluate classes of optimal route queries fast by a unified algorithm on a massive graph. To evaluate a query fast, a graph is partitioned and distances of some optimal paths are materialized. Under such a setting, we found an algorithm, which we called

DiskCP, that can be used to evaluate classes of optimal route queries. To our best knowledge, *DiskCP* is the first disk-based algorithm that can evaluate classes of optimal route queries efficiently. The classes of queries that are evaluated by *DiskCP* are called *constraint preserving* (*CP*). *CP* optimal route queries contain, among others, SP, forbidden edges and nodes, α-autonomy, and *CP* hypothetical weight changes route query classes. *CP* also represent the largest classes of optimal route queries that can be evaluated very fast with distance materialization on a massive graph. We then turn our attention to the optimization of the proposed algorithm. With this method, a query is evaluated in three steps: *sketch graph pruning*, *skeleton path finding* and *skeleton edge filling*. There is not much optimization can be done in the last phase, and techniques were found to speed up the evaluation process for the first two phases. For sketch graph pruning, we generalized existing pruning algorithms [2] to the *CP* optimal query classes. Moreover, we improved the running time by employing the BFS, instead of an exhaustive search. In finding a skeleton path, several optimization techniques were proposed to reduce the execution time and I/O accesses. Experiments were conducted and these techniques have been shown to be effective.

To evaluate *DiskCP*, we implemented a main-memory ($MMCP_{BF}$) and a disk-based ($DiskCP_{BF}$) *CP* route query evaluation algorithms with no data materialization. Three optimal route query classes were chosen in the test, and they are SP, forbidden edges and forbidden nodes. To investigate the scalability of these disk-based algorithms, two data sets were chosen: a relatively small graph called CT and a larger graph named East5. We showed empirically that the proposed *DiskCP* algorithm outperforms, both in term of execution time and I/O accesses, the main-memory and the disk-based algorithms by a wide margin. When respect to CT data set, the execution time of *DiskCP* ranges from 18% to 31% that of required by $MMCP_{BF}$. When compared with $DiskCP_{BF}$, the running time (I/O accesses) ranges from 8% (5%) to 12% (19%) of that required by $DiskCP_{BF}$. For East5 data set, the execution time (I/O accesses) of *DiskCP* is about 1.5% (3%) to 4% (10%) of that needed by $DiskCP_{BF}$. This shows that *DiskCP* is desirable and scalable in evaluating *CP* optimal route queries. This also implies that distance materialization is essential in fast *CP* route query evaluation, especially when the graph is large.

The *CP* optimal query classes can be evaluated fast is due some pre-computation is done on a graph so that this information is used to speed up a route query evaluation. It is an open question to see if and how non-*CP* query classes can be evaluated in a similar fashion. This work requires a graph to be static and the query class to be known in advance so that pre-processing can be performed on the query class. Currently, we are investigating on how to relax these restrictions in fast route query evaluation.

Acknowledgement

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] Chan, E.P.F. and Lim, H., *Evaluation and Optimization of Shortest Path Queries. VLDB Journal* (16:3), July 2007, pp.343-369.

[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*, Atlanta, Georgia, pp.160-166, November 2001.

[4] Guo, L. and Matt, I, "Search Space Reduction in QoS Routing," *Computer Networks 41(1)*, pp. 73-88, 2003.

[5] Li, F., Cheng, D., Hadjieleftheriou, M., Kollios, G. and Teng, S-H, "On Trip Planning Queries in Spatial Databases," *Proceedings of the 9th International Symposium on Spatial and Temporal Databases*, pp. 273-290.

[6] Korkmaz, T., and Krunz, M., "Multi-Constrained Optimal Path Selection," *Proceedings of INFOCOM 2001*, pp.834-843.

[7] 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.

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

[9] Juttner, A., Szviatovszki, B., Mecs, I., and Rajko, Z., "Lagrange Relaxation Based Method for the QoS Routing Problem, *Proceedings of INFOCOM 2001*, pp.859-868.

[10] Papadias, D., Zhang, J., Mamoulis, N. and Tao, Y., "Query Porcessing in Spatial Network Databases," *Proceedings of VLDB*, 2003, pp. 802-813.

[11] Terrovitis, M., Bakiras, S., Papadias, D., and Mouratidis, K., "Constraint Shortest Path Computation", *Proceedings of the 9th International Symposium on Spatial and Temporal Databases*, pp. 181-199.

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

[13] 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*, pp. 20-35, L.A., CA, July 2001.