Finding Shortest Paths in Large Network Systems

Edward P.F. Chan & Ning Zhang
Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada N2L 3G1
epfchan@sdb.uwaterloo.ca
nzhang@hopper.uwaterloo.ca
http://sdb.uwaterloo.ca

May 7, 2001

Abstract

This paper describes a disk-based algorithm for finding shortest paths in a large network system. It employs a strategy of processing the network piece by piece and is based on new algorithms for graph partitioning and for finding shortest paths that overcome the problem of existing approaches. To show that it is scalable to large network systems and is adaptable to different computing environment, seven states in Tiger/Line files are extracted as test cases and are experimented on machines with different configurations. The running time for finding the shortest path depends primarily on the power of the underlying systems. Moreover, to run the algorithm optimally, the memory requirement is not large, even for a very large network system such as the road system in several states in Tiger/Line files. To evaluate its performance, New Mexico state road system is used as the test case, and is compared with Dijkstra's algorithm. The average running time of the proposed algorithm is, in the worst case, about two and a half times slower than that of Dijkstra's algorithm; provided that in Dijkstra's algorithm, the whole graph can be fit into main memory and is already loaded in advance. If the I/O time for loading the whole graph is counted, the proposed algorithm is faster in essentially all cases.

Keywords Shortest Path, Large Graph, Graph Partitioning, Disk-based Algorithm, Route Queries, Spatial Query Processing.

1 Introduction

In Geographical Information Systems (GIS), shortest path queries are one of the most useful and most frequently asked queries. In combinatorics, the shortest path problem on general graphs has already been well-studied. For example, Dijkstra's algorithm is widely used and actually very fast when using heap data structures for priority queues [2]. Even faster algorithms are developed for graphs that have special constraints on their edge weights. For example, Cherkassky, Goldberg, and Radzik developed algorithms based on multi-level buckets [1]. The

constraint of the algorithm is that the weights of the edges must be integers. With this algorithm, the time spent on searching is 1/2 to 1/3 that of Dijkstra's algorithm. However, one assumption common to all the above algorithms is that the whole graph can be stored in main memory. If the graph is too large, the algorithms cannot handle it.

Recently, algorithms have been proposed to address this particular problem. The basic idea is to use divide-and-conquer to break a large graph into small ones, then deal with the small chunks systematically, and at last combine the solutions together. Some papers deal with the partitioning algorithms and the optimality of the solution [6, 7]. Some try to balance between the I/O operations and computation time [8].

Existing disk-based shortest path algorithms are not satisfactory, especially when the graph is large such as road systems in Tiger/Line files. In this paper, new disk-based algorithms are proposed on graph partitioning and for finding the shortest paths. A large network is processed incrementally, both in pre-processing and in querying phase. To allow a large network be processed piece by piece, a swappable data structure called virtual hashtable is employed. A virtual hashtable allows parts of a large object to be main memory resident and are loaded on demand. To show that the proposed method is a truly effective disk-based shortest path algorithm, seven states in Tiger/Line files are extracted as test cases and are experimented on machines with different configurations. The experiment shows that the running time depends largely on the power of the underlying systems. Moreover, to run the algorithm optimally, the memory requirement is not large, even for a very large network system such as the road system in several states in Tiger/Line files. To evaluate its performance, New Mexica state road system is used as the test case, and is compared with Dijkstra's algorithm. The average running time of the proposed algorithm ranges from about the same to two and a half times slower than that of Dijkstra's algorithm; provided that in Dijkstra's algorithm, the whole graph can be fit into main memory and is already loaded in advance. If, for each query, the I/O time for loading the whole graph is counted, the proposed algorithm runs faster in essentially all cases.

In Section 2, we define some basic notation. In Section 3, we survey related work and highlight our contribution. In Section 4, we describe the proposed method. 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. A *fragment* is a connected sub-graph such that an edge connecting two nodes in a fragment iff the edge connecting the two nodes in the original graph. A fragment should be small enough to be main memory resident. A *partition* of a graph is a collection of fragments. A node is a *boundary* node if it belongs to more than one fragment. In Example 1, there are six fragments in the graph. Each fragment is a sub-graph of the original graph and when the six fragments are merged together, we get the original graph. Nodes 23 and 24, for instance, are boundary nodes, and the edge $(23, 24)$ is also shared by the two fragments. In fact, in our implementation, if a pair of boundary nodes appear in more than one fragment and if there is an edge between the pair, the edge appears in only one fragment. For all practical purposes, overlapping between fragments should be minimized.

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

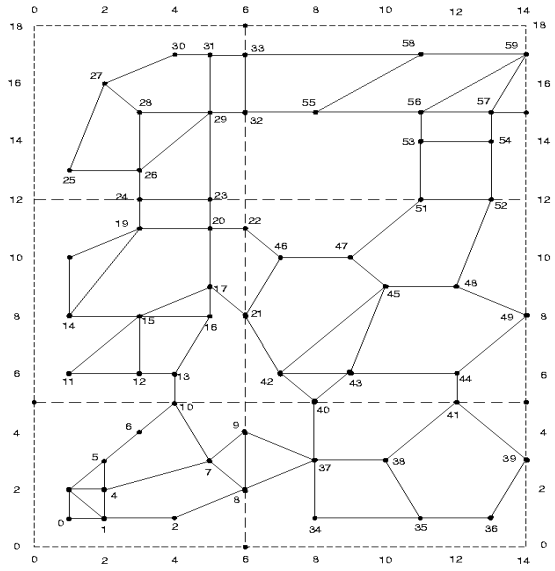

Figure 1: A Graph

The *super graph* of a partition can be thought of as a graph consisting of one complete sub-graph (a clique) for each fragment. The nodes of the super graph are the boundary nodes in the fragment. An edge weight in a super graph is the shortest distance in the corresponding fragment that contains the two boundary nodes, or infinity if no paths connecting them.

3 Related Work

In this section, two disk-based shortest path algorithms are surveyed. The first is proposed by Jing, Huang and Rundensteiner, which is later studied by Shekhar, Fetterer and Goyal [7, 4, 8, 6]. They use a hierarchical approach for shortest path query evaluation. The second algorithm is given by Hutchinson, Maheshwari, and Zeh which use an external-memory extension of a planar graph separator algorithm [5] to partition a large graph and employ external-memory rooted trees for shortest path query evaluation [3, 9]. Both approaches have two phases: pre-processing and querying.

3.1 Hierarchical Graphs

In this method, a graph is first partitioned into fragments, and then boundary nodes are pushed to the second level to form a super graph [7, 8] . All-pair shortest paths among the boundary nodes in the same fragment are also computed, and the corresponding edges are added to the super graph. If the super graph is still too large, it is divided further into fragments and a third level super graph is generated. This process continues until the top-level super graph is small enough to be main memory resident. This whole set of super graphs and the ground level graph is called a *hierarchical graph*. All-pair shortest paths pre-computations at each non-ground level are stored in routing tables.

3.1.1 Pre-processing Phase: Spatial Partitioning Cluster (SPC)

The main idea behind this algorithm is to read in a subgraph and tries to partition it into a set of square-like blocks, where each block corresponds to a page [7]. The graph is processed in a plane-sweep manner, say from left to right. The algorithm works like this:

1. Sort all edges by the x-coordinates of their origin nodes.
2. The sweeping process stops periodically to sort the edges swept since the last stoppage and sort them by y-coordinates of their origin nodes.
3. After each y-sort, the y-sorted edges are grouped into pages and form part of the final partition.

To achieve the objective that a subgraph corresponds to a collection of square-like blocks, the algorithm needs to determine when to stop sweeping and start y-sorting. Their heuristic is: it maintains a temporary block table to store the sorted edges so far, and keeps track of three parameters: dx_i indicates the difference between the minimum and maximum x-coordinate values of the origin nodes in the block table, dy_i indicates the difference between the minimum and maximum y-coordinate values of the origin nodes in the block table, p indicates how many pages of edges have been written to the temporary block table. Based on these values, a stoppage condition is derived [7].

In sum, their heuristic seems to work fine when the graph resembles a square or a cycle. However, when the map is in some peculiar shapes, such as a strip whose x-value range is much greater or less than that of the y-value, the partition based on SPC may not be satisfactory.

After partitioning the ground level graph, one finds the boundary nodes for each fragment and computes the all-pair shortest paths among boundary nodes, then the hierarchical graph is generated bottom-up. A super graph (non-ground level graph) in the hierarchical graph consists of only the boundary nodes of the next lower level graph (super or ground level graph), and there is an edge in the super graph if and only if the two nodes are in the same fragment. The weight of the edge is the shortest distance, with respect to the fragment in which they are in, from one node to the other.

3.1.2 Querying Phase

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 ($s \rightsquigarrow BN(S)$) and those from all boundary nodes in D to d ($BN(D) \rightsquigarrow d$).
2. Compute all possible combinations of $s \rightsquigarrow BN(S) \rightsquigarrow BN(D) \rightsquigarrow d$, and find the minimum one.

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

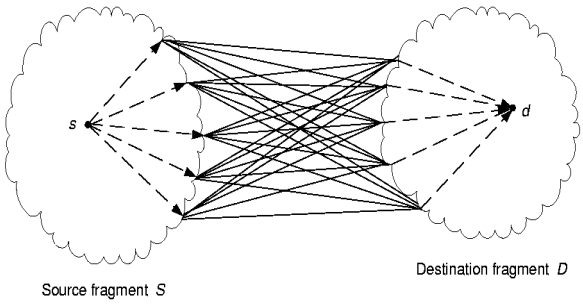

Figure 2: Exhaustive Comparing Algorithm

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 requires 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 $SP(u,v)$ in (*) 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 complexity of exhaustive comparing algorithm on multilevel hierarchical architecture is $O(n^{k-1})$. When k is larger than 3, the asymptotic complexity of the exhaustive comparing algorithm is worse than Dijkstra's algorithm, except that it can cope with larger graphs. However, if $b = O(n)$, then this method may not be effective.

The hierarchical graph is likely to have at least 3 levels. For example, the (relatively small) road system of Connecticut, the number of nodes in the ground level graph is about 160,000, and the number of edges is about 190,000. With the graph partitioning algorithm in Section 4 and by setting the maximum number of edges in a fragment to 15000, the number of boundary nodes is about 1200 for 15 fragments. The size of the boundary nodes in each fragment varies from 120 to 290. The total number of edges in the super graph at level 2 is about 190,000, approximately the same size of the ground level graph.

3.2 Rooted Tree Method

The partitioning algorithm of this approach [3, 9] is an external-memory extension of the Lipton and Tarjan's planar separator algorithm [5]. 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. To see why is this case, the examination of what is stored in the rooted tree and how the shortest path algorithm works is necessary.

3.2.1 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 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 has more than one node, it is recursively divided and should have its own separator S_i, otherwise the node itself is the separator.

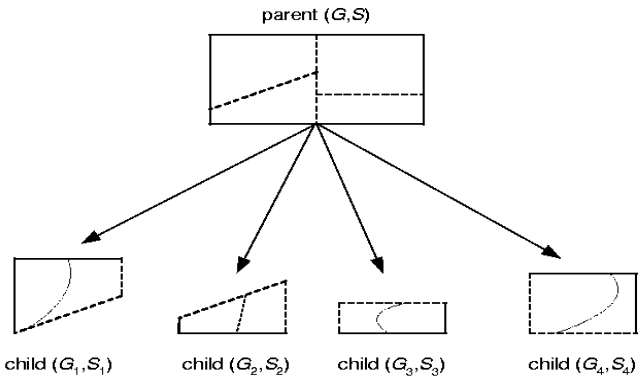

Figure 3: Rooted Tree (dotted lines are separators)

Let us look at an example. A graph G is first partitioned into two connected components A and B using planar separator C. If A or B is not small enough, it is partitioned recursively. At a certain stage, the partitioning algorithm pauses and all the connected components generated since the last pause form one level of the rooted tree. For example, in Figure 3, the parent graph G was first divided into two connected components, then the two components are divided further into four connected components. These four components (G_1, G_2, G_3, 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 G_i is not small enough, they are recusively divided. Note that the children of a graph G do not include nodes of the separator S. Therefore, this guarantees that every node appears in exactly one node in the rooted tree. The tree satisfies the two conditions:

1. At each level, only separators are stored.

2. Each leaf node contains only one node.

3.2.2 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 ancestor, i.e. $B(u, v) = \bigcup S_i, (i \leq l)$. It can be proved that the shortest path between u and v must pass through at least one node in $B(u, v)$. Intuitively, this is correct since if at least one of u or v is in S_l (one node is another's

ancestor), the claim is obviously true. If otherwise u and v are not in S_l, then u and v must be the interior nodes of G_l. The nodes of $B(u,v) - S_l$ are the boundary nodes of connected components that encompass the connected component G_l. Therefore, the shortest path must pass through either some node in S_l or some node in $B(u,v) - S_l$.

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: for each boundary node b in the $B(s,d)$, find the minimum value of $dist(s,b) + dist(b,d)$. 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 [9] 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. This will be too much computation and the storage requirement will be too great. For example, in California road system, there are about 1.6 million nodes. Then there will more than 1 billion shortest path computations and more than 1 billions shortest distances stored *just* for the boundary nodes in the root of the tree. Thus to materialize every shortest path from an interior node to a boundary node requires a lot of computations and storage. This is not feasible for any reasonable large graph.

4 A Disk-based Shortest Path Algorithm

Existing approaches do not seem to work well for large graphs such as road systems in a state in Tiger/Line files. To find a truly effective disk-based algorithm, we use different partitioning algorithm, hierarchical scheme, shortest path querying algorithm and disk-based structure. Our contributions lie in the following aspects:

- A partitioning algorithm based on BFS and spatial index. The advantages of this algorithm are that it is simpler, effective and can be easily extended to certain semantics.
- A 2-level hierarchical graph instead of multi-level (≥ 3) hierarchies are used.
- A disk-based shortest path algorithm that is similar to Dijkstra's algorithm is used, rather than exhaustive comparative algorithm used by the existing approaches.
- I/O is optimized by clustering the results of pre-computation, and storing them as swappable objects which are loaded into memory as needed.

The cause of inefficiency of existing disk-based shortest path algorithms is due to the exhaustive comparative approach. The proposed algorithm is based on Dijkstra's algorithm and thus, unlike existing approaches, searching is incremental and is proportional to the length of a shortest path. We first give a brief description of Dijkstra's algorithm. We then describe the proposed method. Like existing approaches, the proposed method has a pre-processing phase and a querying phase.

4.1 Dijkstra's algorithm

Dijkstra's shortest path algorithm performs a search on a graph from the source node s in iterations. In the subsequent discussion, a node in a graph is said to be *closed* if the shortest distance from s is known. A node is *open* if the shortest distance from s is not known, but a (candidate shortest) distance may be associated with it. In addition to normal properties such as coordinate, a node has three properties: (a) *dist* stores the current shortest distance computed so far for the node from source; (b) *closed* is a boolean indicating if the node is closed; (c) *pred* which records the predecessor in the shortest path if the node is part of it. The *pred* is used to construct the shortest path at the end. The function $w(u,v)$ is the *weight* of the edge (u, v).

Algorithm Dijkstra(*Graph* G=**{**N, E**}**, *Node* s, *Node* d)**:** Find the shortest path from s to d in a connected graph G. *queue.dequeue*() removes and returns a node in *queue* that has the minimum *dist* value.

(1) $s.dist$=0, $s.closed$ = *false* and *queue.enqueue*(s);
(2) for all nodes $n \in N-\{s\}$, $n.dist$=$+\infty$, $n.closed$ = *false*;
(3) *prev* = *null*;
(4) while (*queue* is nonempty)
(5) v= *queue.dequeue*();
(6) v. *pred* = *prev*;
(7) if (v == d) then the shortest path is found and return;
(8) *prev* = v;
(9) $v.closed$ = *true*;
(10) for each non-closed adjacent node u to v do
(11) $u.dist$ = *min*($u.dist$, $v.dist$+$w(v,u)$);
(12) *queue.enqueue*(u);
(13) end /*for*/
(14) end /*while*/

In each iteration, an open node with the minimum distance is selected and closed. Then shortest distances are updated for all open neighbors of the newly closed node. This process is called *relaxation*. The distances of open nodes can be maintained in a priority queue, which is usually implemented by a heap. After the shortest path is found, it can be constructed backward via *pred* from the destination node.

4.2 Virtual Hashtables

To make the algorithm adaptable to different computing environment and to facilitate its scalability, a disk-based structure called *virtual hashtable* is derived. A virtual hashtable is used to swap in and out of data from memory onto the disk. A virtual hashtable is similar to a hashtable; entries are accessed via an identifier. However, not all entries in a virtual hashtable are main memory resident. The application specifies the *maximum* number n of entries in a virtual hashtable that can be main memory resident. The virtual structure ensures that at most n entries are in memory and the well-known LRU replacement scheme is used in swapping in and out of entries onto the disk. A virtual hashtable is useful when a large object such as a graph is only partially main memory resident all the times during a program execution. In our implementation, it is used in pre-processing as well as in querying phase.

4.3 Pre-Processing Phase

The input graph is assumed to be huge, pre-processing of the input is required. The graph is first partitioned into fragments. Some additional pre-processing such as constructing a distance database is also performed. Distance databases are introduced in Section 4.3.2.

4.3.1 Graph Partitioning Algorithm

The input to the graph partitioning algorithm is a general graph, which may or may not be planar. The output is a set of fragments. The resulting partition is stored on a disk and is called a *fragment database*. Throughout the discussion of this partitioning algorithm, it is assumed that only part of the graph can be loaded into main memory at any time, while the rest of the graph is available on the disk.

No matter what a graph is (planar or non-planar) or where it is stored, the graph must somehow be traversed in order to partition it. A good candidate of a graph traversing algorithm is the breadth-first search (BFS), which is also used in the Lipton and Tarjan's planar separator algorithm [5]. The difference from their approach is that the breadth-first spanning tree is not constructed for the whole graph in our algorithm. Rather the algorithm frequently pauses traversing when a condition is satisfied. The traversed nodes and edges are then extracted to form a fragment and are saved to the disk. The objects in a fragment are removed from main memory to save space for other untraversed part of the graph to be loaded. This heuristic cannot guarantee the optimality of the node separator (i.e. no more than $2\sqrt{2n}$ boundary nodes) as done by the Lipton and Tarjan's algorithm, but the payoff is its simplicity to implement, the ability to process a large graph piece by piece and its capability to partition non-planar graphs. Another nice property of this algorithm is that it can be easily extended to accommodate particular semantics. For example, the graph can be partitioned such that some special type of street blocks, say interstate highway, can only be on the boundary of fragments. In this case, the pause condition is that all the next boundary nodes ready for BFS exploring are nodes corresponding to interstate highway intersections. A more realistic case is to combine many conditions together to form a pause function. Whenever the function returns true, the exploring process pauses and saves the result.

Before fragments are generated, a graph need to be broken up into smaller pieces. This is facilitated with a spatial index such as an R-tree variant. The area covered by the graph is first divided into grids so that all objects in a grid can be main memory resident. Each grid is denoted by a *minimum bounding rectangle* (*mbr*). The way the grids are derived could depend on the size and shape of the graph, and perhaps its density. Objects in each grid are stored as a unit on the disk which can then be read into main memory by the partitioning algorithm. The initial grid collections as well as the result of grid partitioning are stored as virtual hashtables.

Algorithm GraphPartition(g,min,max):

Input: g is a graph indexed by an R-tree, min and max are the preferred minimum and maximum number of edges in a fragment, respectively.

Output: A fragment database.

Method:

(1) Divide the graph area into grids, each of which is denoted by an *mbr*.
(2) For each grid, query the R-tree with its *mbr* to retrieve all intersecting nodes and edges. For each such grid collection, store it as a graph on the disk.
(3) Arbitrarily select a node from a grid collection and perform a BFS with the node as the root. The search could span one or more grids. Grid collections are read into memory whenever they are needed. Nodes and edges selected in the search are removed from the

collection and from memory to form a fragment. The search terminates either it reaches the min size or no more object can be found. The fragment resulted is output to disk.

(4) Repeat step (3) until g is empty.

(5) Merge tiny fragments produced in steps (3) and (4) so that their size is between min and max.

With regard to time complexity, the predominate factor is step (3). In this step, a BFS is performed during which it requires to look up adjacent nodes in a fragment. The time complexity of GraphPartition can be shown to be $O(n \log max + m)$, where n and m are the number of nodes and edges in the graph and max is the maximum number of edges in a fragment.

4.3.2 Distance Matrix and Distance Database

Conceptually, one can apply Dijkstra's algorithm to the graph, by reading in fragment by fragment, once it has been partitioned. To minimize computation and IO accesses, shortest distances between boundary nodes within fragments are pre-computed. For each fragment in the partition that has a boundary node, a *distance matrix* is created to contain the shortest distance between pairs of boundary nodes in the fragment. 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 ground level graph and a super graph. A super graph is implicitly represented with a distance database.

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

The disk-based shortest path algorithm 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 $matrixDB$. Both the fragment database and distance database are stored as virtual hashtables.

Consider conceptually a graph g obtained by merging the fragments S, D and the super graph. If applying Dijkstra's algorithm to g with s and d, we will obtain the "skeleton" shortest path. A "skeleton" path consists of a sequence of nodes from S, D and the super graph. The edge between two nodes in a super graph is a "skeleton" edge. A "skeleton" edge denotes the shortest path in the corresponding fragment for the two boundary nodes. Once the "skeleton" shortest path is obtained, the "skeleton" edges are replaced with the shortest paths in the corresponding fragments. This is the main idea behind the proposed shortest path algorithm. The problem is that the super graph is huge and therefore the graph cannot be constructed explicitly. Since Dijkstra's algorithm is incremental in nature, it is not required to have the whole super graph main memory resident. Instead, whenever a boundary node is closed in the relaxation process, the distance between two adjacent boundary nodes as well as information on these nodes, are read from the disk, if they are not already in cache.

Let us look at how a super graph is represented implicitly during program execution. A *boundary set* is the set of boundary nodes that are shared by two fragments. The edges, more precisely the edge weights, in a super graph are implicitly represented by a distance database. Edges in a super graph are grouped according to the fragment in which they are in and each group is denoted by a distance matrix in the distance database. On the other hand, nodes in a super graph are grouped as boundary sets. Organizing nodes as boundary sets, as opposed to fragments, would reduce memory requirement during execution. Each boundary set is represented by a *distance vector* which stores node information and properties such as distance computed so far from source. The distance vectors form a *distance vector database*. The distance database and distance vector database are each implemented as a virtual hashtable. Entries in these virtual hashtables are distance vectors and distance matrices, respectively.

Distance vectors as well as distance matrices are read into memory whenever they are needed and is done automatically with a virtual hashtable.

Initially all nodes (except s) in S, D and all boundary nodes have their distances from s set to infinity. The distance of s is 0 and is the only open node. Initially only graph $S+D$, the graph obtained by merging S and D, is in main memory. During the execution, whenever a node in g is closed, the relaxation process in Dijkstra's algorithm is applied to every adjacent boundary node in g. Since the distance matrices and distance vectors are stored in virtual hashtables, this process may incur I/O operations if they are not in cache yet. In each iteration, a non-closed node with the minimum shortest distance is closed. The node could be a node in S or D or a boundary node in other fragment. The relaxation and closing process keeps going until the destination node is closed.

After this we get a "skeleton" of the shortest path. The "skeleton" path consists of three parts: the shortest path from s to a boundary node b_s in S, the shortest path from a boundary node b_d in D to d, and a sequence of boundary nodes on the shortest path between b_s and b_d. The first two parts are complete sub-paths in the resulting shortest path, but the third part may have missing interior nodes between each pair of boundary nodes. The last step of this algorithm is to "fill in" the missing nodes by looking up the intermediate fragments from the fragment database and applying Dijkstra's algorithm to these fragments with a source and a destination.

Let us first look at the data structures for implementing this algorithm. All entries in a queue below is a of the form <*key*, *value*> pair, where *key* is used to determine the priority among entries in the queue. All queues have the following operations: *min()* returns the minimal key. *minValue()* returns the value of the minimal key. For *min()* and *minValue()*, only non-closed nodes are considered, if the queue is either *dv* or *SDQueue*; *dequeue()* removes the entry with the minimal key and returns the entry's value; *updateValue(key, value)* which updates the *key* if the *value* exists and adds the *key-value* pair otherwise. All these functions return *null* if the queue is empty.

1. *matrixDB*: The distance database. A virtual hashtable that stores distance matrix for each fragment in the partition. The function *get(fid)* returns the distance matrix for the boundary nodes in the fragment identified by *fid*. This has been constructed in the pre-processing phase.

2. *matrix*: For each pair of boundary nodes in a fragment, *get* returns the shortest distance between them with respect to the fragment involved.

3. *SDQueue*: A queue. It contains nodes in the source and destination fragments. The key is the *dist* property of the node while the value is the node.

4. *dv*: A distance vector, one for each boundary set, is identified with the boundary set id. It is implemented as a queue. Each node in the boundary set has exactly one entry in the queue. The value of an entry is a node in the boundary set while its key is the node's *dist*.

5. *BNQueue*: A queue. Each boundary set has precisely one entry in this queue. An entry is <d, $BSId$>, where $BSId$ denotes a boundary set while d is the shortest distance computed so far among all nodes in the set from a source. Whenever the shortest distance of a node in a distance vector is changed, the corresponding entry is updated.

6. *dvDB*: Distance vector database. A virtual hashtable that contains all distance vectors. One entry in the hashtable for each distance vector. The function *get(bsid)* returns the distance vector for the boundary set *bsid* while *find(n)* returns the boundary node stored in some distance vector with the same coordinate as a node n.

Algorithm diskSP(s,d,S,D, F, M): s and d are the source and destination nodes, S and D are the fragments in which s and d are in respectively, F is the fragment database, M is the distance database.

```
(1)  initialization();
(2)  while(SDQueue not empty or BNQueue.min() != +∞)
(3)      a = SDQueue.min();
(4)      b = BNQueue.min();
(5)      if ((a is null) or (b<a))
(6)          bv= dvDB.get(BNQueue.minValue()).minValue();
(7)          relaxBV(bv,b,M,dvDB,BNQueue);
(8)          if (bv is in D)
(9)              /* sd.find return the node in sd that has the same coordinate as the boundary node bv.*/
(10)             bv' = sd.find(bv);
(11)             set bv' properties to those in bv and return if bv' == d;
(12)             relax(sd, bv', SDQueue);
(13)         end /*if */
(14)     else
(15)         v = SDQueue.dequeue();
(16)         if (v==d) fillSP(F,d) & return;
(17)         if (v is a boundary node)
(18)             v' = dvDB.find(v);
(19)             set v' properties to those in v;
(20)             relaxBV(v',a,M,dvDB,BNQueue);
(21)         end /* if */
(22)         relax(sd,v,SDQueue);
(23)     end /*if*/
(24) end /*while*/
```

Algorithm initialization():

(1) Merge graphs S and D to form graph sd.
For the source s, set *dist* to 0.
For all other nodes in sd, set *dist* to $+\infty$.
All nodes in sd has the *closed* property set to *false*.
(2) Initialize *SDQueue*: add the source s to the queue.
(3) Initialize *BNQueue*: for each boundary set, add an entry ($+\infty$, *id*), where *id* is the boundary set id.
(4) Initialize *dvDB*: for each boundary set, add an entry (*id*, *dv*), where *id* is the boundary set id and *dv* is a distance vector for boundary set *id*.
For each node (except s, if it is in the set) in a distance vector, set its *dist* to $+\infty$, and *closed* property to *false*.

Algorithm relaxBV(v, d, *matrixDB*, *dvDB*, *BNQueue*): v is a boundary node in some distance vector in *dvDV* and is the next closed boundary node. d is the shortest distance from a source to v. Relax all adjacent boundary nodes to v.

(1) Let *list* be a list of adjacent boundary set ids to v, including that containing v.
(2) for each *id* in *list* do
(3) $dv = dvDB.get(id)$;
(4) OldMin = $dv.min()$;

(5) *updateDv*(*dv*,*v*,*d*,*matrixDB.get(id)*);
(6) if (*dv.min()*!=OldMin) *BNQueue.updateValue*(*dv.min*(), *id*);
(7) end /*for*/
(8) *v.closed* = *true*;
(9) *dv* = *dvDB.get*(*Bid*), where *Bid* is the id for the boundary set containing *v*;
(10) *BNQueue.updateValue*(*dv.min*(), *Bid*);

Algorithm relax(*g*, *v*, *SDQueue*): The node *v* is a node in graph *g* and is the next closed node. Relax all adjacent nodes to *v* in *g*.

(1) *v.closed* = *true*;
(2) for each non-closed node *u* in *g* adjacent to *v* do
(3) $d = w(v, u)+v.dist$;
(4) if ($u.dist > d$)
(5) $u.dist = d$;
(6) *u.pred* = *v*;
(7) *SDQueue.updateValue*(*d*, *u*) ;
(8) end/*if*/
(9) end /*for*/

Algorithm updateDv(*dv*, *v*, *d*, *matrix*): *dv* is a distance vector for a boundary set that is adjacent to the node *v* and *v* is the next closed node. *d* is the shortest distance of *v* from a source. Update the distance from the source for all non-closed nodes in *dv*. *matrix* is the distance matrix for the fragment in which *v* and *dv* are in.

(1) for each non-closed node *o* in *dv* do
(2) *dist* = *matrix.get*(*v*, *o*)+*d*;
(3) if (*o.dist*> *dist*)
(4) *o.dist*=*dist*;
(5) *o.pred*=*v*;
(6) end /* if */
(7) end /*for*/

Algorithm fillSP(F, dest): Given a destination node *dest* and a fragment database *F*, traverse *.pred* to get the shortest path in reverse order.

In our implementation, *SDQueue* and *BNQueue* are binary heaps while *distance vectors* are Fibonacci heaps. The major operations in *diskSP* are applying Dijkstra's algorithm to the abstract graph *g*. So the proposed algorithm has the same time complexity as Dijkstra's algorithm.

5 Performance Evaluation

In the pre-processing phase, the graph is partitioned into fragments. We will look at the result of the partitioning. In the querying phase, we will concentrate on the memory requirements and compare the running time of the proposed algorithm with Dijkstra's algorithm.

5.1 Data and Environment

Road systems of seven states are chosen from Tiger/Line files to form four test cases: Connecticut (CT), New Mexico (NM), California (CA) and the eastern five states (East5, which is composed of Connecticut, Massachusetts, New Jersey, New York, and Pennsylvania). The four test cases range from a small graph (CT) to a large graph (East5). The data on these test cases are summarized in Table 1.

State	Number of edges	Number of nodes	Size (MB)
CT	197861	161595	42
NM	603786	487814	130
CA	2062863	1669288	225
East5	3169730	2507774	338

Table 1: Test Cases Statistics

The primary system for testing is a dual-processor system with two Pentium III 933MHz CPU's and 1 GB RDRAM, with 16KB level 1 cache and 256KB level 2 cache. The hard disk is Ultra 160 SCSI drive. To show that the proposed algorithm is adaptable to various computing environment, a second less powerful system is used. The secondary system is a Pentium III 500MHz processor with 384MB SDRAM and IDE hard drives. The operating system in both machines is Microsoft Windows 2000 Server SP1. We use Sun Java 1.3 HotSpot Client VM (build 1.3.0-C mixed mode). The road systems of California and Connecticut are tested on both machines with several hundreds of randomly generated pairs of nodes. The only difference is the running time, which depends largely on the power of CPU and the speed of the disk.

For the rest of this section, the test results obtained are from running on the primary system.

5.2 Pre-processing: Partitioning Algorithm

In this experiment, we set the minimum and maximum number of nodes in fragments to be 15,000 and 20,000. The statistic values are shown in Table 2.

State	Number of fragments	Number of boundary sets	Number of boundary nodes	Average number of boundary nodes per boundary set	matrixDB Size (MB)	dvDB Size (MB)
CT	10	18	1003	55	1.78	0.066
NM	32	76	3216	42	5.85	1.39
CA	108	265	11705	44	25.82	5.65
East5	164	402	19434	48	41.63	11.9

Table 2: Partitioning Statistics

From the table, the average numbers of boundary nodes in boundary sets are around 40 to 55. The average number of boundary sets per fragment is from 3.6 to 4.9. The average number of edges in fragments ranges from 19,045 to 19,952. The number of boundary nodes is about $(n/170)$, where n is the number of edges in a graph. The extra storage is relatively small when compared to the database size. The time to perform partitioning is proportional to the size of the test case.

5.3 Querying Phase

As virtual hashtables are used in the querying phase, one factor that affects the algorithm performance is the maximum number of entries specified for a virtual hashtable. The optimal number of entries dictates the memory requirement and thus the scalability of the algorithm. This issue is investigated in Section 5.3.1. To show that the proposed algorithm is effective, it is compared with Dijkstra's algorithm with respect to the execution time. The result is summarized in Section 5.3.2.

5.3.1 Memory Requirements

We want to investigate how the memory size affects the performance of the algorithm. Since East5 is the largest, it is chosen as the test case. Distance vector and distance matrix are the two most frequently used virtual structures in querying phase. By varying the cache sizes for distance vector database and for distance database, the execution time for finding a "skeleton" path is collected. We do not count the running time of *FillSP* algorithm since that depends on how far apart the destination from the source node. The cache sizes for distance vector database are 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30, 40, 50, 60, 70 while for distance database are 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 25, 30, 40. For each combination, a shortest path query is issued and the execution time is recorded.

We first investigate the effects of the cache sizes to the running time when the JVM is sufficiently large. Figure 4 shows the relation among the size of caches for distance vector database and distance database and the running time with 500MB JVM. We then repeat the test with 60MB JVM. The shape remains more or less the same when the JVM is 60MB. The only difference is that the execution time increases slightly with smaller virtual memory.

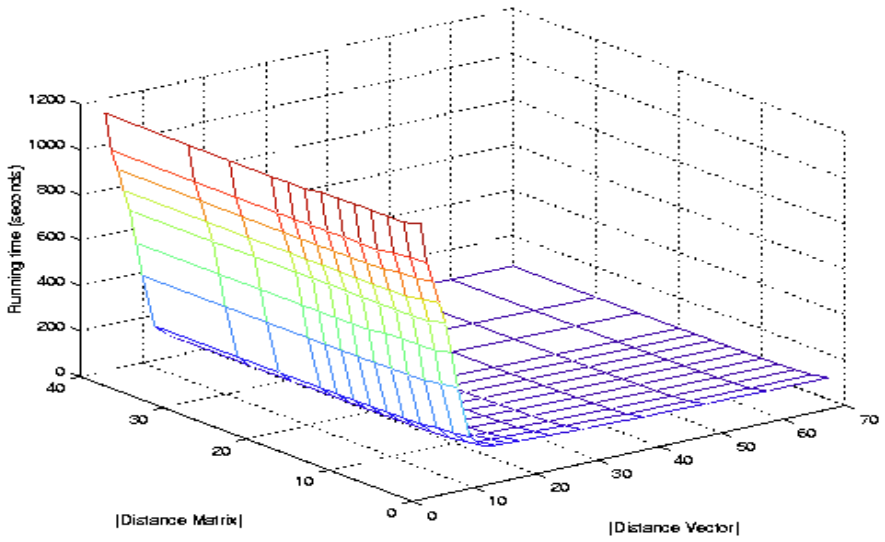

Figure 4: Running time with various cache sizes for distance vector database and distance database with 500MB JVM

The cache size for distance vector database greatly affects the running time if it is less than 18. For the distance database, the cache size does not affect too much of the performance, so long as it is greater than 5. This can be seen from the cross section of the figure along $|DistanceVector|$ axis and $|DistanceMatrix|$ axis, which is shown in Figure 5.

Figure 6 shows the running time for different amount of memory assigned to JVM when the cache sizes of distance vector database and distance database are set to 18 and 10, respectively.

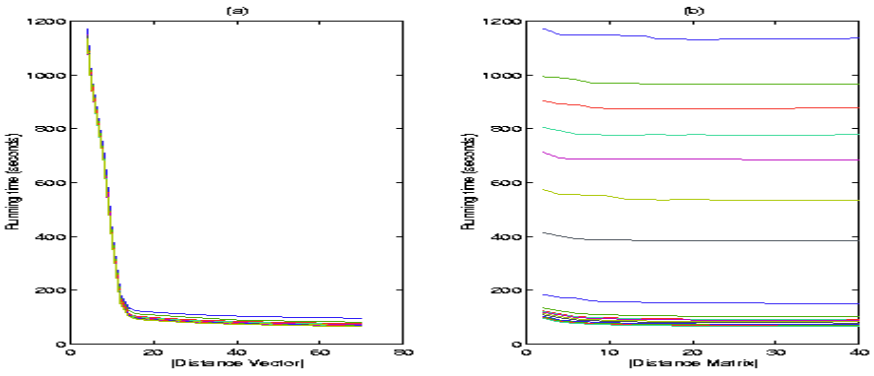

Figure 5: The effects of cache size of distance vector and distance matrix (a) running time for different cache size of distance vectors (b) running time for different cache size of distance matrix

The memory size ranges from 26MB up to 476MB. Each point in the diagram is a query execution time and the same query is used for all memory sizes. Even with 60MB of JVM and with a large graph East5, the running time is very decent and is close to optimal. This shows that finding the "skeleton" path does not require a large amount of virtual memory and thus the proposed algorithm is effective and scalable to very large graphs.

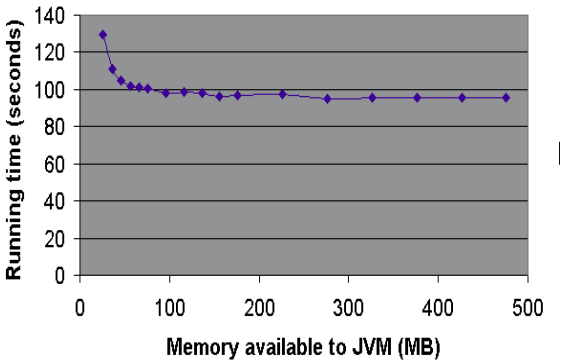

Figure 6: Running time for different JVM sizes with $|matrixDB| = 10$, $|dvDB| = 18$

5.3.2 Execution Time and IO Comparison

To see how the disk-based algorithm compares to Dijkstra's algorithm, 1000 pairs of nodes are generated for NM. Since Dijkstra's algorithm requires the graph to be memory resident, the JVM sizes are set to 500MB in both cases. The cache sizes for distance database and distance vector database are set to 10 and 20, respectively. The test case NM is chosen because it is of resonable size for 1000 test cases and the whole graph can be main memory resident. Java has a maximum virtual space of 999MB. For East5, the graph is too big to be main memory resident. For each pair of source and destination, the shortest path is computed and statistics on execution time and IO are collected. Thus the result for *diskSP* includes the invocation of *FillSP* algorithm.

To facilitate the presentation, the queries are grouped together according to the length of the shortest path. These groups represent source and destination are very close to very far apart. The average is used to denote the group value. For execution time, the ratio of *diskSP*

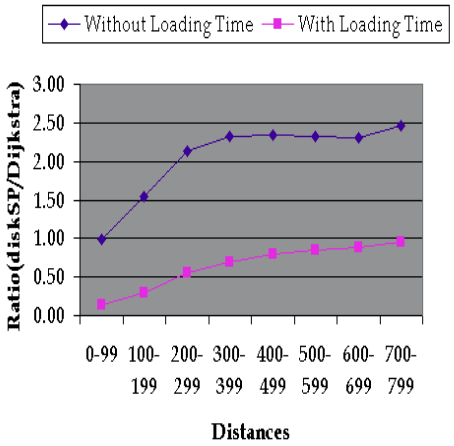

Figure 7: Execution time Comparison

to Dijkstra is given. The execution time for queries with *diskSP* ranges from several seconds up to 100 seconds. In Dijkstra's algorithm, it is required that the graph to be read into memory.

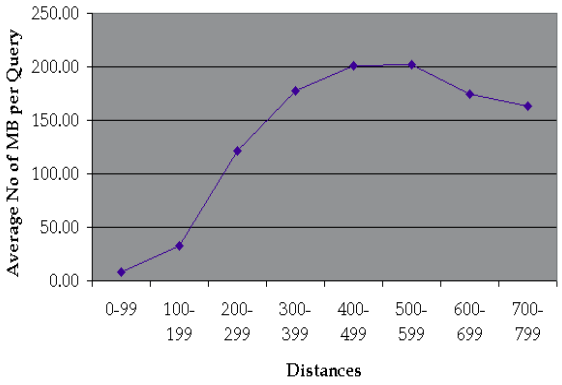

Figure 8: IO for diskSP

Figure 7 compares the two algorithms for cases when the loading time is and is not included in Dijkstra's algorithm. From the figure, algorithm *diskSP* requires as much time as Dijkstra and up to two and a half times slower if the graph is loaded in advance. However if, for each query, the loading time is taken into account, algorithm *diskSP* is faster in essentially all cases. Algorithm *diskSP* performance deteriorates with the length of the shortest path, since more data are accessed in algorithm *fillSP*. This is confirmed by Figure 8.

6 Conclusion

Memory-based shortest path algorithms are well-studied and many theoretical and empirical results have derived. However, disk-based shortest path algorithms are not studied very well. Some of the previous works have built some theoretical models and have gotten empirical results from prototypes, but none of them has shown that their algorithms are practical to large graphs such as the road system of a state in Tiger/Line file. This paper proposes a disk-based algorithm working like Dijkstra's algorithm and has gotten promising empirical results from the real road

systems.

An important parameter of the proposed disk-based algorithm is the cache size of virtual data structures. Experiments show that if the cache size is less than a certain value for distance vector database, the performance deteriorates very fast. This is because of the locality of relaxation operations in the shortest path algorithm. We also show that to obtain optimal performance of the algorithm, the virtual memory required is not large, even for very large network systems. This suggests that the algorithm is scalable to graphs of very large sizes.

Experiment shows that the running time of the proposed disk-based shortest path algorithm ranges from about the same to two and a half times slower than that of Dijkstra's algorithm, provided that in Dijkstra's algorithm, the whole map can be fit into main memory and is already loaded in advance. If, for each query, the I/O time for loading the whole graph is counted, the proposed disk-based shortest path algorithm is faster in essentially all cases. This is because we make use of the pre-computation information and it is well-clustered on the disk. This shows that the proposed method is a truly effective disk-based shortest path algorithm. The performance of the algorithm can further be improved and is a topic that is currently under investigation.

Acknowledgement

The authors wish to thank the financial support of Bell University Laboratories, and of the Natural Sciences and Engineering Research Council of Canada.

References

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

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

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

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

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

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

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

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

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