Path Queries for Transportation Networks: Dynamic Reordering and Sliding Window Paging Techniques *

Yun-Wu Huang
University of Michigan
ywh@eecs.umich.edu

Ning Jing[†]
Changsha Institute of Technology
ningjing@pdns.nudt.edu.cn

Elke A. Rundensteiner[‡]
Worcester Polytechnic Institute
rundenst@cs.wpi.edu

Abstract

It is well-known that standard relational database engines are not equipped to efficiently process path finding queries required by diverse applications, such as Geographic Information Systems (GIS), Intelligent Transportation Systems (ITS), etc. In this paper, we explore the development of special-purpose disk-based techniques for supporting the processing of path queries on transportation networks. To process high-throughput path queries, our solution materializes *path views* and employs several innovative *path view* refreshing strategies. One key technique is to perform *dynamic reordering* of link tuples in breadth-first order based on recent I/O activities. This reclustering approach trades extra I/O overhead required by reordering for the performance gains achievable by path search on approximately topologically ordered link tuples. We also introduce a *sliding window paging policy* that further reduces page misses. We present experiments conducted in evaluating our approach and in comparing it with previously published disk-based methods using both real GIS data (city road maps) and synthetic grid graphs. Our results show that our new approach outperforms the alternative approaches significantly for highly cyclic graphs such as GIS maps.

1 Introduction

1.1 Motivation: ITS Shortest Path Finding

Path finding is an essential feature for many Geographic Information Systems including navigation systems, urban planning, emergency response and traveler information systems. This paper in particular investigates the route guidance services offered by Intelligence Transportation Systems (ITS). In centralized beacon-based route guidance approach, each ITS vehicle in the system submits path finding queries through roadside beacons to the Traffic Management Centers (TMCs) and, in real-time, receives instructions of where to go next. The widespread deployment of the beacons guarantees that the guided vehicles travel along the shortest paths that are continuously computed en route, based on dynamic traffic conditions. This paper focuses on path query optimization strategies for an ITS database that supports such path queries.

For applications that require such high throughput path query support, conducting a full-scale path search for each path query submitted is not feasible. To satisfy the time constraint for a potentially large numbers of path queries submitted during high-traffic hours, we propose the TMCs create and maintain a materialized view of best routes, called the *path view* [9]. Thus, path queries can be serviced by direct lookup of the the *path view* rather than computing paths on-the-fly. The path query turnaround time is minimized at the cost of periodical *path view* computation. To maintain the correctness of the *path view*, which is crucial due to dynamically changing traffic patterns, recomputation of the *path view* must be efficient so that it can be refreshed frequently.

To support path queries at the database system level, we must consider the situation when the availability of the DBMS system memory buffer is limited due to simultaneous tasks from multiple users. In addition, the DBMS engine may have to concurrently process I/O-intensive constraints of complex path queries, such as "Find the shortest path from Ann Arbor to Detroit that does not go through flooded areas" — thus requiring major portions of the shared system buffer for this *spatial constraint* computation. Last but not least the potential size of the *path views* can be prohibitively large for large graphs. All of this points towards the requirement of disk-based solutions. This paper presents an innovative approach that efficiently computes *path views* in a disk-based environment by exploiting unique properties of the transportation networks modeled by ITS maps.

1.2 Previous Approaches: ITS Versus General Path Problems

Recent database transitive closure (TC) research [2, 3, 12, 13, 14] is dominated by solving general path problems with

*This work was supported in part by the University of Michigan ITS Research Center of Excellence grant (DTFH61-93-X-00017-Sub) sponsored by the U.S. Dept. of Transportation and by the Michigan Dept. of Transportation.

[†]This work was performed while the author was visiting the University of Michigan.

[‡]This work was performed while the author was a faculty member of the University of Michigan.

the focus on providing recursive query capabilities for general databases. Such research thus typically starts with the "reachability" problem and treats the shortest path problem as a side product. This is evident in the two techniques adopted by many of the recent approaches for database TC proposed in the literature, namely node collapsing and topological ordering. Both are not effective in solving the shortest path problem for cyclic graphs, denoted as SP_{cyclic} in this paper. Node collapsing techniques [19] cannot be applied to solve SP_{cyclic} problem because shortest paths between two arbitrary nodes in the strongly connected component are not identical as is the case for "reachability". Topological ordering insures that reachability TC computed for the descendent nodes can be shared by their ancestor nodes without additional computation. For SP_{cyclic}, the ancestor relation is mostly bi-directional. The topological ordering technique only preserves the ancestor relation in one direction; it does not guarantee good paging behavior when the computation traverses the other direction.

We argue that SP_{cyclic} is a unique yet important problem in that solutions designed to solve the general path problems generally have bad performance when applied to SP_{cyclic}. Note that experiments for SP_{cyclic} were done with graphs of only up to 200 nodes in [13] and under 300 nodes in [2], and for both the performance was already deteriorating dramatically. This is clearly not sufficient for ITS applications where graph sizes are typically above one thousand nodes.

1.3 Our Proposed Solution: The *BSR* Approach

Our solution is based on the observation that ITS graphs that model city road maps have several unique properties. They are highly cyclic, have low outdegree (mostly 2 to 5), a large number of nodes (up to tens of thousands), and high locality[1]. In this paper, we propose an integrated ITS path view solution, called BSR[2], that effectively exploits the above properties of transportation networks. Our solution is designed to address the problems of the target application and as our experimental results demonstrate, this new approach indeed is a significant improvement over previous disk-based techniques for ITS graphs.

One key technique of our proposed solution is to, based on recent I/O activities, perform *dynamic reordering* of link tuples in breadth-first order. This reclustering approach trades off extra I/O overhead required by reordering with the performance gains achievable by path search on approximately breadth-first ordered link tuples. In order to effectively deal with a limited main memory buffer size, we introduce a *sliding window* paging policy that further reduces page misses. In this paper we propose and study different criteria for reordering that include historical trends and fixed intervals.

We also present detailed analyses and experimental results evaluating our proposed *BSR* system, and comparing it against alternative approaches. These alternatives include a *BFS* algorithm with only preordering, a reverse Depth First Search *DFS* algorithm that is similar to the shortest path algorithm, *Path_BTC* [13], and the *Warshall* algorithm. We experimented with a real GIS map and synthetic grid maps of sizes up to 1,600 nodes. Our results show that *BSR* outperforms the other three algorithms dramatically on these highly cyclic maps. The superior performance of *BSR* on smaller buffer sizes demonstrates that our solution remains effective in a shared-buffer database environment with multiple tasks running at the same time. It is obvious that larger buffer sizes can handle even larger maps. We believe our solution is an important improvement over known results considering that previous research only tested cyclic maps with sizes up to 300 nodes for the SP_{cyclic} problem.

1.4 Paper Outline

The paper is organized as follows. We describe the related work and background in Section 2. In Section 3, we present the proposed *path view* solution. We describe our testbed in Section 4 and present the experimental results in Section 5. In Section 6, we conclude this paper and provide an outlook of our future work.

2 Related Work and Background

2.1 Related Work

Recently proposed disk-based transitive closure algorithms [2, 3, 4, 6, 7, 11, 12, 13, 18] in the literature generally focused on the reachability problem, and then extend the solutions to solve other path problems. Some of them [6, 7, 18] could not be extended to solve the shortest path problem for cyclic graphs (SP_{cyclic}) and only a few of them addressed the shortest path problem for cyclic graphs [2, 13].

The experimental results reported in [2, 13] showed that their algorithms are not suited to solve SP_{cyclic} because their performance deteriorated dramatically when the graph sizes go beyond 300 nodes. Our experimental results (Section 5) show that our proposed *BSR* has a better performance and can handle much larger graphs for solving SP_{cyclic} problems.

Shekar et al. in [16] focused on the compute-on-demand approach by comparing several typical route computation algorithms, such as *Dijkstra*, breadth-first search and A^* heuristic search. By implementing these algorithms directly on relational DBMS products, they found that the response time to process a shortest path query can be up to 20 minutes for a GIS street map of about 1000 nodes.

Agrawal and Jagadish [1] presented (using simulation) a path encoding structure that has an acceptable storage overhead – compared to maintaining all possible paths. A shortest path version of it can be adopted to materialize *encoded path views* [9]. The *BSR* with minor modification can be considered an I/O efficient algorithm that creates disk-based encoding structures for ITS graphs.

Our previous work [9] compared path query processing between the *path views* and the single-pair path search approach

[1] We define high locality maps by when the two end nodes of all links are geographically closely located.

[2] BSR stands for Breadth-first search using Sliding windows and dynamic Reordering techniques.

approaches. We found that that the *path view* approach is more efficient for high throughput path finding requirements as found in ITS systems. In the context of an in-memory approach, we also explored hierarchical optimization strategies - both optimal [15] and heuristic ones [10]. In [8], we focus on compute-on-demand path finding with constraints be evaluating various clustering optimizations.

2.2 ITS Graph Characteristics

Graphs that model road transportation networks are a subset of the general graphs defined above. They exhibit some unique characteristics that can be exploited by a path computation system such as the *BSR* presented in this paper. In general, such ITS graphs

1) have a large numbers of nodes, up to tens of thousand.

2) have uniformly low outdegree (usually 2 - 5).

3) exhibit a high locality, which means the two end nodes of each link are usually geographically closely located.

4) are highly cyclic (indeed, most roads are two-directional, thus resulting in numerous cycles of 2 links in each direction, and any node is typically reachable from any other node).

5) are near-planar with a few exceptions such as overpasses.

2.3 Data Representation

In this paper, we assume the topological information of a graph is modeled by a a relation (table) that stores all the links in the graph. The data representation of this *link relation* is an ordered set of tuples of the form $< source, destination, weight >$. We assume that tuples of the same *source* are clustered together in the *link relation*. Links of the same *source* are ordered by *destination*. We define the handle of a node i, $\&i$, to be the reference to the *link relation* that points to the first tuple whose *source* is i. The data representation for the *path view* is also an ordered set of tuples of the form $< source, destination, weight >$, with $weight$ corresponding to the cost of the shortest path from the *source* to *destination*. The tuples in the *path view* are sorted with *source* as primary key and *destination* as secondary key. In practice, the next hop or the previous hop can be added to each *path view* tuple to facilitate the retrieval of the next hop or the entire shortest path in a trivial manner.

3 The *BSR* - An Integrated Solution

The *BSR* materializes the *path views* for a given graph by computing a single-source *path view* for every node in the graph using a single-source *BFS* shortest path algorithm. The *BSR* deploys two optimizations: the *sliding window* paging method and the *dynamic reordering* technique. Note that other single-source shortest path algorithms such as the $Dijkstra$ or A^* can also be used but their I/O behavior dictated by our *sliding window* optimization would show the same patterns as that of the *BFS* algorithm.

3.1 The Single-Source BFS Shortest Path Algorithm

The single-source breadth-first search shortest path algorithm expands nodes in the graph in breadth-first traversal order. To expand a node means to traverse the links from this node to all its children nodes. In our implementation, the traversal is done by retrieving the link tuples whose $source$ is the current expansion node. If the tuples are not in main memory, then they must be read in from the *link relation* that is stored on disk. If the main memory is full, a paging mechanism takes action to make room for them.

For cyclic graphs, the children nodes of the expansion node may or may not have been expanded before. We say a child node of the expansion node is good for future expansion and called it a *prospect* node if one of the two following conditions is true:

1. The child node has not been expanded before.
2. The child node has been expanded, but the current expansion results in a shorter path weight from the *root* node to this child node.

3.2 The *Sliding Window* Method: Optimization 1

We now propose an *intra-* single-source *BFS* optimization, called the *sliding window*. The basic idea of the *sliding window* method is to load a contiguous portion of the *link relation*, called the window, into main memory, and to conduct breadth-first search within the window. Any expansion thread that references tuples outside of the window is delayed until all *intra*-window expansion threads have completed. Figure 1 illustrates the projection of the window in the *link relation* onto the main memory. If the size of the main memory is larger than the size of the *link relation*, the window contains the entire *link relation* and the *BSR* approach degenerates into a main memory shortest path *BFS* algorithm. It is the reverse case that is more complex and more I/O intensive, and on which we shall focus our discussion in this paper.

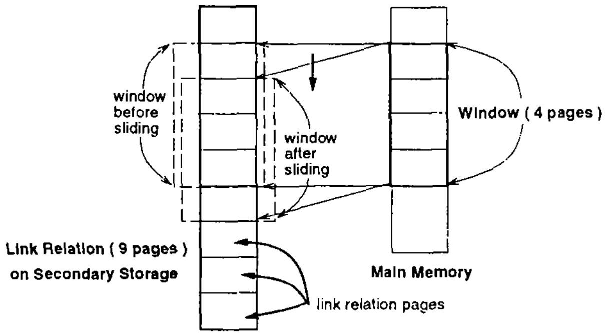

Figure 1: The Sliding Window.

The *link relation* is segmented into pages and the size of the window is a multiple of that of a page. During path search expansions, all threads that traverse outside of the window are suspended. When all *intra*-window expansion threads are exhausted, we slide the window down one page on the *link relation*, wrapping around if necessary. Once a new window is brought up, the suspended expansion threads that reference to the tuples in the new page are recovered and their expansions restarted. The method terminates when there is no intra-window expansion and suspended threads left.

3.3 Dynamic Link Reordering: Optimization 2

The *dynamic reordering* is an optimization that exploits the characteristics of ITS maps. Based on the ITS graph characteristics outlined in Section 2.2, typical ITS maps resemble grid graphs. By studying grid graphs, we observe that, for any link tuple $< i, j, w >$ in the BFS-ordered *link relation* of a grid graph, $\&i$ and $\&j$ will not be far apart. To illustrate this, consider *BFS* as a priority search based on the accumulated path label d where d is the aggregation of all link labels of the search path, and label for each link is set to 1. The expansion starts with $d = 0$ which is the root node itself, $d = 1$ which are the children nodes of the root node, then $d = 2$ which are the nodes reachable by the root node in two hops, etc. Suppose the root node is the center node of the grid graph, such process creates waves of diamond-shaped outward expansions, depicted in Figure 2 using dotted lines. We denote C_i as the number of nodes expanded for the wave of expansion such that $d = i$. Let the grid graph be a $m \times m$ graph where $n = m^2$ and n is the number of nodes in the graph, we derive $\forall i, C_i \leq 2 \times (m - 1)$.

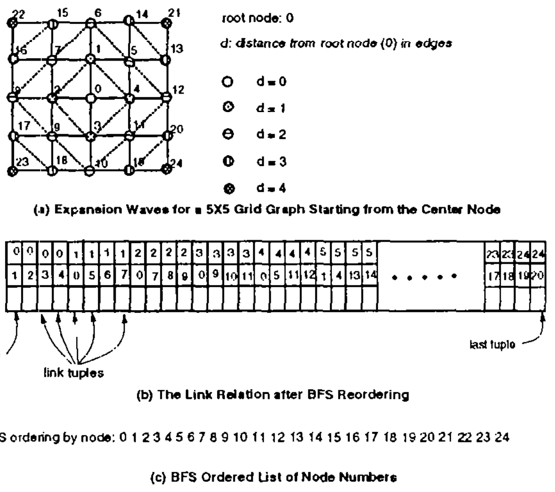
root node: 0
d: distance from root node (0) in edges
d=0
d=1
d=2
d=3
d=4
(a) Expansion Waves for a 5X5 Grid Graph Starting from the Center Node
source
destination
weight
first tuple
link tuples
last tuple
(b) The Link Relation after BFS Reordering
BFS ordering by node: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
(c) BFS Ordered List of Node Numbers

Figure 2: Distance Between Two End Nodes of A Link for A Grid Graph.

Although the *BFS* reordering of the *link relation* involves link tuples as shown in Figure 2 (b), we can conceptualize the result of the reordering being a list of source nodes (Figure 2 (c)). We conclude that, within the *BFS* ordering of the source nodes, node i and node j are separated by no more than $2 \times (m - 1)$ nodes if $< i, j, w >$ is a tuple in the *link relation*. For example, in Figure 2 (c), the two end nodes of a link that are farthest apart in the *BFS* ordering are node 11 and node 20, which are separated by 8 nodes. The maximum C_i are $C_2 = C_3 = 2 \times (5 - 1) = 8$. Note that the maximum C_i is actually smaller if the root node is a corner node because the expansion waves form lines of 45°, maximizing at the diagonal line where $C_{m-1} = m$. Comparing with the possible maximum of $m \times m - 2$ nodes between the two nodes of a link in unordered *link relation*, the distance of $2 \times (m - 1)$ represents a good *expansion locality*.

For node i such that $\&i$ is close to the top of the ordered *link relation*, fresh expansions[3] in the single-source *path view* go mostly in one direction (left figure of Figure 3). Because each expansion reaches only nearby nodes, the single-source *path view* computation for node i is likely to display a good *expansion locality* in the *link relation*. With sufficient window size that covers this *expansion locality*, we expect that the single-source *path view* computation exhibits excellent paging behavior.

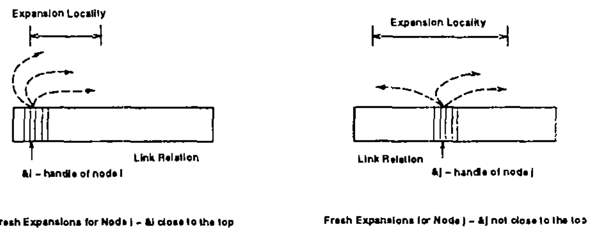

Figure 3: Expansion Locality: Node Near Top Versus Node in the Middle.

This is not true for nodes that are away from the top of the ordered *link relation* (right figure of Figure 3), because the fresh expansions go both ways. The *expansion locality* in the *link relation* grows as nodes are being expanded. It is likely that the expansion locality may eventually exceed the size of the main memory, causing deteriorated I/O or worse, thrashing.

Based on the above observation, we propose to reorder the *link relation* periodically so that the single-source *path view* is computed for nodes that are always at the top of the *link relation*. As a result, the overall I/O in computing the *path view* for all nodes is optimized at the cost of processing reorderings repeatedly. The extreme case would be to reorder the *link relation* for every node whose single-source *path view* is to be computed. This may not be feasible because the cost of reordering could out-weigh the saving in I/O caused by the reordering. Therefore, there exists a balance between too many reorderings which increase the reordering cost and too few reorderings which would result in deterioration of I/O for *path view* computation.

For this reason, we propose a *dynamic reordering* mechanism that reorders the *link relation* based on recent historical I/O performance. The *dynamic ordering* mechanism determines the frequency and timing of reordering using two parameters, K and R. K captures the number of nodes whose single-source *path views* were last computed. R is the *reorder threshold*. The average I/O for the last K single-source

[3] A fresh expansion expands a node for the first time.

computations, called *Recent Average I/O*, is recalculated after each new single-source *path view* is computed. We also keep the average I/O of the first 10 single-source *path view* computations[4] immediately after the last reordering. We call this the *Initial Average I/O*. The *Initial Average I/O* corresponds to the average I/O for nodes that are on the topmost part in the ordered *link relation*. Computations for these nodes should exhibit excellent paging behavior because their fresh expansions are mostly one way.

Our *dynamic reordering* strategy can be described as follows. *Dynamic reordering* happens when the *Recent Average I/O* is worst than the *InitialAverage I/O* by a percentage of R or more. Therefore, the smaller R is for a given K, the more frequently the reordering will be performed. The smaller K is for a given R the more sensitive the reordering is to the variations of recent I/O. For example, if K is 10, reordering is performed only when the average I/O of the last 10 nodes is worse than the *Initial Average I/O* by R. A small number of spikes in I/O in the last 10 nodes will not raise the average high enough to invoke reordering. If K is set to 2, any spike in I/O in the last 2 nodes is likely to increase the average enough to cause reordering.

3.4 The *BSR = BFS + Sliding Window + Dynamic Reordering*

The *BSR* first preorders the *link relation* in the *BFS* order. Then it conducts single-source *BFS* shortest path algorithm with the *sliding window* optimization for each node in the graph exactly once. The processing order of the iteration is determined by each node's offset in the *link relation*, in ascending order. In other words, the *BSR* traverses the *BFS* ordered *link relation* from the root node downwards. The *BSR* takes two parameters, K and R, and uses them to determine when reordering of the *link relation* needs to be performed. In Section 5, we will present experiments evaluating the effectiveness of different value combinations for parameters K and R. After reordering, it starts from the top of the *link relation* downwards and runs the single-source *BFS* algorithm for the next *source* node whose single-source *path views* have not yet been computed. The *BSR* continues the reorder-compute cycle until the single-source *path views* for all nodes in the graph are computed.

3.5 Main Memory Management of the *BSR*

The main memory is segmented into pages of equal size. We use P to denote the number of total pages in main memory. The P pages are further separated into two regions, the input region (L) and the output region (T). The *sliding window* corresponds to the input region which is used to store the link tuples from the *link relation*. The output region is used to store the single-source *path view* during its computation. When the computation of a single-source *path view* is completed, it is written out to disk from the output region which is reused for next single-source computation. Each single-source *path view* is written out exactly once. Figure 4 illustrates the two regions of the main memory.

[4] 10 is an arbitrary number. From our experiments, the I/O of the single-source *path view* computations for the first 20 - 50 nodes are always equally good. So 10 is a safe number.

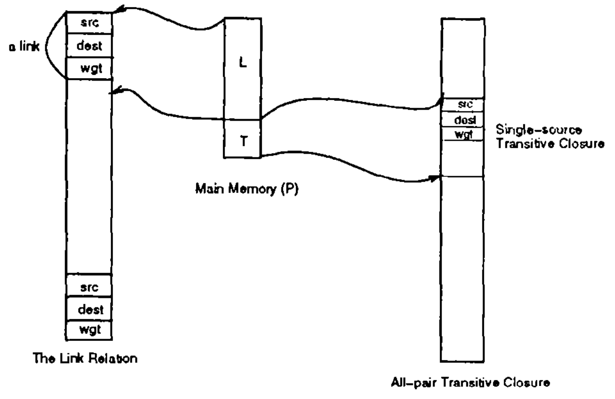

Figure 4: Main Memory Paging Configuration.

4 Testbed Setup

4.1 Alternative Approaches for Comparison

For comparison purposes, we implemented three alternative algorithms, the *BFS* shortest path algorithm, the reverse *DFS* algorithm and the *Warshall* algorithm. We call them *BFSSP*, *RDFS*, and *Warshall*, respectively. All three algorithms follow the same data representation presented in Section 2 and use LRU as their paging policy.

The *BFSSP* algorithm first preorders the *link relation* in *BFS* order by the *source* nodes. It then runs the single-source *BFS* algorithm iteratively for each node in the graph. The order of the iteration is determined by each node's handle in the *link relation* in ascending order. This algorithm therefore corresponds to a partial *BSR* without the *sliding window* and *dynamic reordering* optimizations.

The *RDFS* algorithm simulates Ioannidis et al.'s recently proposed *Path_BTC* [13]. In *Path_BTC*, pages are segmented into blocks to minimize internal fragmentation and to reduce page I/O. In our context, the number of descendents in each descendent set is equal to the number of nodes in the graph because the GIS map we tested is strongly connected in whole. Therefore, it suffices to treat the single-source *path view* and the descendent set as one structure in the *RDFS* algorithm.

The *Warshall* algorithm is a direct implementation of the well-known *Warshall* matrix-based shortest path algorithm based on secondary storage [21].

4.2 Implementation Environment

All algorithms are implemented on an IBM-RS6000 workstation that runs the Unix (AIX) operating system. The *source* and *destination* fields of the link tuples and the shortest path tuples are each a 2-byte short integer and *weight* is a 4-byte integer, making each tuple 8 bytes in length (both same as in [13]). The *path view* is sorted first by the *source* field, then by by the *destination* field. The *path view* enumerates all possible *source-destination* pairs. If the shortest path between a *source-destination* pair does not exist, its *weight* field remains ∞ after the computation of the *path view* is complete. This however will not happen for the GIS map in our experiments because it is strongly connected.

The majority of the main memory is segmented into pages, each 2K bytes in length (same as in [13]). To present a consistent benchmark comparison, we use the same buffer sizes used in [13], namely 10 and 20 pages. In addition to the paged memory, we keep several data structures in main memory. These are two bitmaps for marking status, an array of half-bytes to store the outdegree of each node, an array of 2-byte short integers to store the handles in the *link relation* for each node, and an array of 2-byte short integer for queue or stack operations. The length of all these data structures is the total number of nodes in the graph.

We test both synthetic grid graphs and a real map representing the street map of Troy City in Michigan. The grid graphs are created by establishing perfect grid graphs as in Figure 2, but with the outdegree of the randomly selected 5% of the links reduced to 2, 20% reduced to 3, and 10% raised to 5. The percentage of deletion and addition of links simulates the outdegree distribution of the Troy City map.

5 Experimental Results

This section presents the results of our experiments. Because the Unix operating system does its own file block buffering, the elapse time can not serve as an accurate barometer for performance. In our experiments, the performance is evaluated using the simulated number of I/O in pages.

5.1 Experiments on *Dynamic Reordering*

Figure 5 shows the progression of I/O activities between *dynamic reordering* and *no reordering* running the *BSR* on the real Troy City map. We use a 10-page main memory buffer, and set the *dynamic reordering* parameters R to 20 and K to 2. This means the reordering is performed when the average I/O of the two most recent single-source *path view* computations is worse than the average I/O of 10 computations recorded right after the last reordering by 20%. We record the average I/O for every 20 consecutive computations and plot the results in Figure 5.

Figure 5 shows that if no reordering is done, the paging behavior is good only for the first 60 nodes or so. It becomes erratic and tends to go upwards as the computations of the single-source *path view* continues for nodes not at the beginning of the *link relation*. In comparison, the I/O cost remains flat if the *dynamic reordering* mechanism is incorporated.

5.2 Reordering Policy: Dynamic vs. Static vs. No Reordering

We experiment with *BSR* using three reordering policies: *dynamic reordering*, *static reordering*, and *no reordering*. For *dynamic reordering*, we vary K over 2, 5 and 10, and for each K value, we vary R from 5 to 160. For *static reordering*, we choose a parameter N and conduct reordering whenever N single-source *path views* have been computed. We experiment with N ranging from 10 to 100. For each set of experiments, we use three types of graphs: a 900-node grid graph with $P = 10$, a 1600-node grid graph with $P = 20$ and the real map with $P = 20$.

The results of the experiments on shown in Figure 5 the 900-node grid graph, in Figure 7 for the 1590-node real map, and in Figures 8 for the 1600-node grid graph. The horizontal straight lines in all experiments represent the I/O costs if no reordering is conducted.

It is clear from these results of Figure 6 that *dynamic reordering* with $R < 100$ results in best performance in terms of page I/O. For clarity, we combine the dynamic and the static reordering results into one chart although their x-axis values have different meaning. For *dynamic reordering*, the x-axis represents the *reorder threshold* percentage while for *static reordering*, the x-axis stands for constant intervals between reorderings. From Figures 6, 7, and 8, it is evident

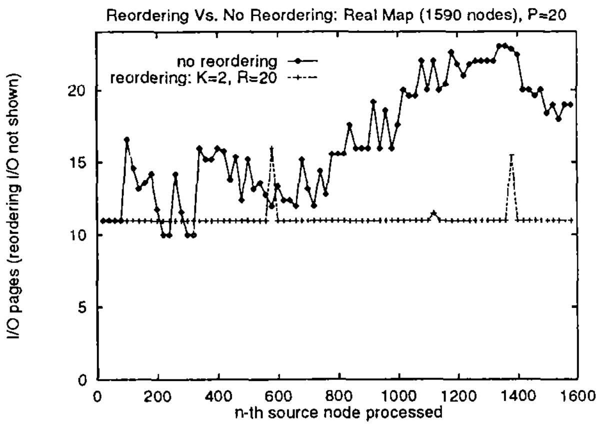

Figure 5: Reordering Vs. No Reordering.

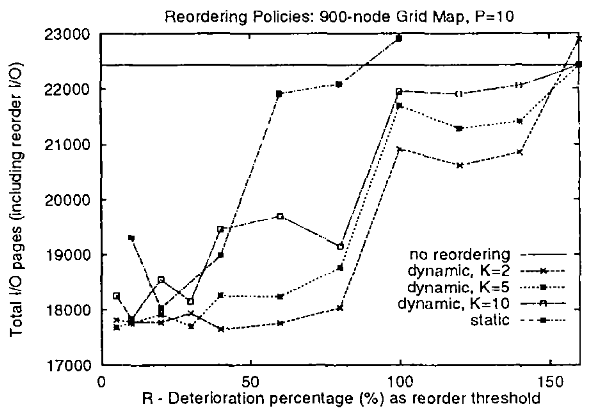

Figure 6: Reordering Policies: 900-node Grid Map.

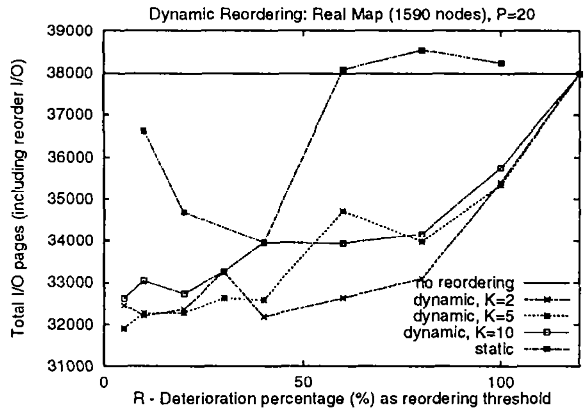

Figure 7: Reordering Policies: 1590-node Real Map.

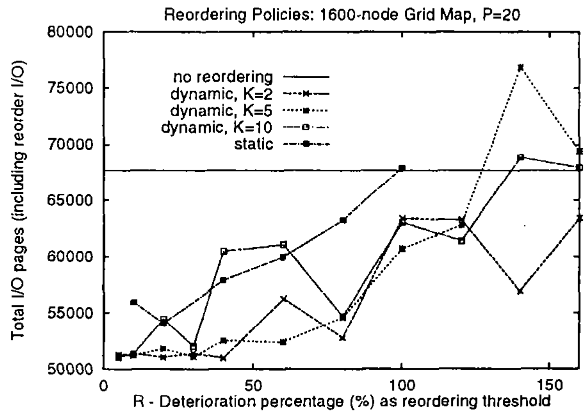

Figure 8: Reordering Policies: 1600-node Grid Map.

that the lowest point of the *dynamic reordering* is lower than any point of the *static reordering* in all cases. We conclude *dynamic reordering* is superior to *static reordering*.

Our results show that the performance of *dynamic reordering* is the best when $R < 40$. This means that more sensitive *dynamic reordering* results in better performance. Furthermore, when $R < 40$, the I/O performances for $K = 2$ and 5 are better than those for $K = 10$. This indicates that reordering performance is better when it is more sensitive to recent I/O performance by deriving *Recent Average I/O* using a small history window (K). All I/O costs in the results include the I/O incurred by the reordering process itself. These sets of experiments confirm that, for GIS maps, sensitive *dynamic reordering* helps reduce I/O in the computation of the *path view*. The overall regulated paging behavior far outweights the additional cost incurred by the reordering processes.

In Figures 9, we pick the winners from *dynamic reordering* and *static reordering*, and compare them with *no reordering*. It is clear from this figure (Figure 9), sensitive *dynamic reordering* dominates *static reordering* and *no reordering*.

	Dynamic Reordering K=2, R=10		Static Reordering N=20		No Reordering	
	I/O	Ratio to Winner	I/O	Ratio to Winner	I/O	Ratio to Winner
900-node Grid P=10	17770	1	18024	1.01	22431	1.26
1225-node Grid P=10	35480	1	36351	1.02	49008	1.38
1600-node Grid P=20	51384	1	54046	1.05	67695	1.32
Real Map (1590) P=20	32202	1	34679	1.08	37992	1.18

Figure 9: Dynamic Reordering Vs. Static Reordering Vs. No Reordering in I/O.

5.3 *BSR* Vs. Alternative Algorithms

We compare the *BSR* against the alternative shortest path algorithms introduced in Section 4, namely the *BFSSP* algorithm, the *RDFS* algorithm, and the *Warshall* algorithm. The experiments are performed using synthetic grid graphs of size 196, 400, 625 and 900, and with 10 pages in main memory. For *BSR*, we set the reordering parameters to $K = 2$ and $R = 10$. The results in page I/O are displayed in Figure 10. The *BSR* clearly dominates others. For graphs of size 196 and 400, the size of the *link relation* is smaller than the input region of the memory, making the *sliding window* method of the *BSR* and the LRU paging policy of the *BFSSP* algorithm irrelevant. The I/O costs of graphs of size 196 and 400, for both the *BSR* and the *BFSSP*, include one read through the *link relation* and one write-out of the entire *path view*. Hence, *BSR* reduces down to the main memory version of the simpler *BFSSP* in this case.

It is reported in [13] that it takes *Path_BTC* about 8000 I/O pages to compute the shortest path transitive closure for a 200-node cyclic graph using 10 2K-byte pages in main memory. Our experiment results in Figure 10 show that *RDFS* needs 2403 I/O pages to compute the shortest path transitive closure for a 196-node grid graph using the same amount of main memory. We don't claim that *RDFS* is better than *Path_BTC*, rather our goal was to design an algorithm that is similar in computational efficiency to the *Path_BTC* algorithm. When the graph size grows from 196 to more than 400 in our experiments, the I/O for *RDFS* increases dramatically. This is caused by the increase in the size and the number of the descendent sets that need to be brought into main memory, and by the fact that the time complexity of *RDFS* is $O(n^3)$. We think the *Path_BTC* algorithm will have

Page I/O	BSR K=2, R=10	BFSSP	RDFS	Warshall
196-node Grid P = 10	199	199	2403	63460
400-node Grid P = 10	806	806	18460	345864
625-node Grid P = 10	4709	47810	102314	927710
900-node Grid P = 10	17770	257110	464594	1987649

Figure 10: Comparison Between Disk-based Algorithms.

a similar increase pattern for large graphs, although [13] does not report any result for graphs of size larger than 200.

6 Conclusion and Future Work

In this paper, we address the problem that standard relational database engines are not efficient in path query processing required by applications such as Intelligent Transportation Systems (ITS) [16]. Our solution is to materialize a *path view* of *all shortest* paths and to perform look-up instead of complete path search from-scratch for incoming path query requests. This allows efficient processing of path queries with a higher throughput. To refresh the path view efficiently, we propose the *BSR* algorithm. *BSR* incorporates several innovative disk-based optimization techniques, namely *sliding window* and *dynamic reordering* techniques. We present experiments conducted in evaluating our approach and in comparing it with existing disk-based methods using real GIS map data and synthetic grid graphs. Our results show while sensitive dynamic reorderings yield best performance, our new approach in general outperforms the alternative approaches proposed in the literature [13, 21] significantly for cyclic graphs such as GIS maps. Therefore, *BSR* can be deployed in centralized ITS route guidance for more efficient *path view* re-computation.

Our future work includes comparing *BSR* with other graph clustering techniques such as min-cut partitioning, topological ordering, spatial partitioning, connectivity clustering [17], etc.

References

[1] Agrawal, R. and Jagadish, H. V., "Materialization and Incremental Update of Path Information", *IEEE 5th Int. Conf. on Data Engineering*, 1989, pp. 374 – 383.

[2] Agrawal, R., Dar, S., and Jagadish, H. V., "Direct Transitive Closure Algorithms: Design and Performance Evaluation," *ACM Transactions on Database Systems*, Vol. 15, No. 3, Sep. 1990, pp. 427 – 458.

[3] Agrawal, R. and Jagadish, H. V., "Hybrid Transitive Closure Algorithms," *Proc. of the 16th VLDB Conf.*, Brisbane, Australia, 1990, pp. 326 – 334.

[4] Bancilhon, F., "Naive Evaluation of Recursively Defined Relations", In *On Knowledge Base Management Systems – Integrating Database and AI systems*, M. Brodie and J, Mylopoulos, Eds., Springer-Verlag, New York, 1985

[5] Dijkstra, E. W. "A Note on Two Problems in Connection with Graphs", *Numer.* March, 1959, pp. 269 – 271.

[6] Ebert, J., "A Sensitive Transitive Closure Algorithm", *Information Processing Letters, 12.* , 1981, pp. 255 – 258.

[7] Eve, J. and Kurki-Suonio, R., "On Computing the Transitive Closure of a Relation", *Acta Informatica, 8.* , 1977, pp. 303 – 314.

[8] Huang, Y.W., Jing, N., and Rundensteiner, E. A., "Effective Graph Clustering for Path Queries in Digital Map Databases", *Proc. 5th Int'l Conf. CIKM*, Nov. 1996.

[9] Huang, Y. W., Jing, N. and Rundensteiner, E., "A Semi-Materialized View Approach for Route Maintenance in Intelligent Vehicle Highway Systems," *The Second ACM Workshop on Geographic Information Systems*, Washington, D.C., Nov. 1994.

[10] Huang, Y.W., Jing, J., and Rundensteiner, E. A., "Hierarchical Path Views: A Model Based on Fragmentation and Transportation Road Types," *ACM Workshop on Geographic Information Systems*, Nov. 1995, pp. 93 - 100.

[11] Ioannidis, Y. E., "On the Computation of the Transitive Closure of Relational Operators," *Proc. 12th Int'l Conf. VLDB*, Aug. 1986, pp. 403 – 411.

[12] Ioannidis, Y. E. and Ramakrishnan, R., "An Efficient Transitive Closure Algorithm," *Proc. 14th Int'l Conf. VLDB*, Aug.-Sep. 1988, pp. 382 – 394.

[13] Ioannidis, Y. E., Ramakrishnan, R., and Winger, L.., "Transitive Closure Algorithms Based on Graph Traversal," *ACM Transactions on Database Systems*, Vol. 18, No. 3, Sep. 1993, pp. 512 – 576.

[14] Jagadish, H. V. and Agrawal, R., "A Study of Transitive Closure as a Recursion Mechanism," *Proc. ACM-SIGMOD*, May 1987, pp. 331 – 344.

[15] Jing, N., Huang, Y.W., and Rundensteiner, E. A., "Hierarchical Optimization of Optimal Path Finding for Transportation Applications" *Proc. 5th Int'l Conf. CIKM*, Nov. 1996.

[16] Shekar, S., Kohli, A. and Coyle, M., "Path Computation Algorithms for Advanced Traveller Information Systems," *IEEE 9th Int. Conf. on Data Engineering*, 1993, pp. 31 – 39.

[17] Shekar, S. and Liu, D.R., "CCAM: A Connectivity-Clustered Access Method for Aggregate Queries on Transportation Networks : A Summary of Results," *IEEE 11th Int. Conf. on Data Engineering*, 1995, pp. 410 – 419.

[18] Schmitz, I., "An Improved Transitive Closure Algorithm," *Computing 30*, 1983, pp. 359 – 371.

[19] Tarjan, R. E., "Depth-First Search and Linear Graph Algorithms," *SIAM J. Computing, 1, 2*, 1972, pp. 146 – 160.

[20] Warren, H.S. "A Modification of Warshall's Algorithm for the Transitive Closure of Binary Relations," *Comm. ACM, 18, 4*, 1975, pp. 218 – 220

[21] Warshall, S. "A Theorem on Boolean Matrices," *JACM, 9, 1*, 1962, pp. 11 – 12