Effective Graph Clustering for Path Queries in Digital Map Databases *

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

Ning Jing†
Changsha Institute of Technology
jning@eecs.umich.edu

Elke A. Rundensteiner
University of Michigan
rundenst@eecs.umich.edu

Abstract

In this paper, we present an experimental evaluation of graph clustering strategies in terms of their effectiveness in optimizing I/O for path query processing in digital map databases. Clustering optimization is attractive because it does not incurs any run-time cost, and is complimentary to many of the existing techniques in path query optimization. We first propose a novel graph clustering technique, called Spatial Partition Clustering (SPC), that creates balanced partitions of links based on the spatial proximity of their origin nodes. We then select three alternative clustering techniques from the literature, namely two-way partitioning, approximately topological clustering, and random clustering, to compare their performance in path query processing with SPC. Experimental evaluation indicates that our SPC performs the best for the high-locality graphs (such as GIS maps), whereas the two-way partitioning approach performs the best for no-locality random graphs.

1 Introduction

1.1 Motivation

Digital map databases are critical components of Geographic Information Systems (GIS) applications such as navigation, route guidance, traveler information systems, fleet management, public transit, and traffic management. An important type of services required by many of the above applications is path query processing in digital map databases. Examples of path queries are:

Q1: ***Find the most energy-efficient path from A to B that does not use toll roads.***

Q2: ***Display all the garages reachable from A in 10 minutes.***

Q3: ***Find the shortest path from A to B that does not pass through areas with altitude > 1000 feet.***

All three path queries have embedded constraints. Namely, the computed path contains no links of *toll* road type for **Q1**; the destination nodes are of *garage* type for **Q2**; and the computed path contains only links that do not traverse areas with *altitude > 1000 feet* for **Q3**.

In order to process path queries such as the above, a map database must model the topological information of the maps as well as maintain the attributes associated with each map element. Typically, the topological information of a road map is decomposed into nodes and links. A node represents an intersection and a link represents a road segment which is one section of a road between two neighboring intersections where traffic flows in one direction.

In this paper, we assume path computation is based on cost measurements that are only associated with links[1], called *link weights*. We also assume that query computation for path queries such as those listed above (**Q1 - Q3**) is based on graph-traversal algorithms[2] which conduct path search computation by traversing from one node to another through their connecting link. Because path search computation is recursive in nature, searching a path means to recursively retrieve link tuples in the *link table* into the main memory buffer for evaluation. Since the size of the *link table* is often larger than the main memory buffer of a given map database, many tuples in the *link table* may need to be retrieved numerous times during the computation of a path on a large digital map. This amounts to tremendous I/O costs required for the computation of path queries.

1.2 Goals

To address this high-cost problem of path query computation, we set the following goals in this paper:

(1) To identify existing clustering techniques suitable for optimizing the I/O behavior for path query processing for digital map databases.

(2) To develop novel clustering techniques that exploit unique properties of digital map databases such as spatial coor-

*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.

†On leave from Changsha Institute of Technology, China, is currently visiting the University of Michigan.

CIKM 96, Rockville MD USA
© 1996 ACM 0-89791-873-8/96/11 ..$3.50

[1] We assume traversing a node is costless and that such costs, if exist, have been factored appropriately into weights of the connecting links.

[2] The *Dijkstra*, *A**, Breadth-First Search, and Depth-First Search algorithms are all examples of popular graph-traversal search algorithms.

dinates or high locality of digital maps.

(3) To implement the selected clustering techniques on a uniform testbed for fair comparison.

(4) To perform experimental evaluation of both existing as well as our proposed clustering techniques to determine their relative effectiveness.

1.3 Effective Clustering: Key to Path Query Processing

The purpose of this paper is to demonstrate that clustering optimization for path query computation can be effective for cyclic graphs such as GIS maps. Clustering is attractive because it does not incur any run-time cost and it requires no auxiliary data structure that demands buffer space. For attributes that do not change frequently, clustering can be done off-line with no dynamic update costs. Furthermore, clustering is at a level lower than many other path query solutions that focus on auxiliary access structures [2, 16] or on algorithmic techniques [1, 3]. Therefore results emerging from the comparative evaluation of our clustering research can be deployed by solutions that do not employ specific link clustering [1, 3, 16].

In this paper, we first present a clustering technique that partitions links in a GIS map based on the spatial proximity of the *origin* nodes of the links. We call it Spatial Partition Clustering (SPC). Because GIS links are short, most nearby links are grouped into the same partition by SPC. Since graph-traversal algorithms exhibit high expansion locality on GIS maps, SPC is expected to optimize I/O incurred during path search. Next, we implement three other alternative clustering strategies, namely approximately topological clustering [2], two-way partition clustering [5], and random clustering. For our benchmark studies of these techniques, we select the *Dijkstra* algorithm [6] because it is one of the most popular and effective single path search algorithms[3] for sparse graphs such as GIS maps.

We conduct experiments using a real GIS map of Ann Arbor (5,596 nodes, 14,033 links) and randomly generated graphs of similar size. For each clustering strategy, we process path queries using the *Dijkstra* algorithm with various buffer sizes and different graph localities[4]. Our results show that random clustering performs the worst in all cases, confirming that graph clustering techniques can be effective in reducing the I/O costs of path query processing. Among the more effective clustering approaches, our spatial partition clustering performs the best for high-locality graphs (such as GIS maps) whereas the two-way partitioning approach is the best for graphs without locality.

The outline of this paper is as follows. Section 2 discusses related work, followed by Section 3 that introduces the proposed spatial partition clustering technique. In Section 4 we present the alternative graph clustering techniques, and in Section 5 we outline our testbed environment. We give the experimental results in Section 6, and conclude the paper in Section 7.

[3] Note that another popular algorithm, A^*, can be viewed as a variation of the *Dijkstra* algorithm with heuristics.

[4] We define that in a graph of high locality, the two end nodes of most links are located closely geographically. For graphs of no locality, such restriction does not apply.

2 Related Work

There are many recent research efforts reported in the literature that focus on minimizing the I/O costs of path computation in a database setting that assumes a main memory I/O buffer of fixed size. Most of such research has focused on pure transitive closure (*tc*) computation [1, 3, 10]. In our previous work, we have explored a hierarchical path view approach which fragments a large graph into smaller subgraphs and pre-computes the path transitive closure for each subgraph [8, 9, 11]. The advantage of such a technique is more efficient computation in both transitive closures of the subgraphs and path search through the hierarchy.

Two potential problems exist in using *tc* pre-computation to answer path queries for digital map databases. First, a single *tc* cannot take various embedded constraints into account. For example, a *tc* computed for path query **Q1** cannot be used to answer path query **Q3**. To answer all path queries with many different embedded constraints, numerous *tc* may need to be computed, each based on a unique embedded constraint. This may not be feasible in practice. Second, some link weights may change very frequently. In order for the *tc* computed based on such weights to remain current, recomputation may need to be conducted very frequently. However, performance results in [1, 10] showed that such techniques are not efficient in computing the shortest path *tc* for cyclic graphs such as GIS maps managed by digital map databases. Recomputation of the shortest path *tc* using such techniques therefore cannot be done frequently, undercutting the correctness of the computed paths.

In [16], a graph indexing technique is proposed to improve paging performance for graph traversal by building an auxiliary structure to predict nodes that are to be accessed in the future. [16] gave a cost model but did not present experimental results on real GIS maps.

Topological clustering techniques have been previously proposed to reduce the path query processing I/O costs [2, 4, 13]. Pure topological clustering [4, 13] does not apply to cyclic graphs such as GIS maps. In [2], an approximately topological clustering was proposed that handles cyclic graphs using heuristics which break cycles to decompose a graph into acyclic subgraphs. It is suggested that such a technique may not be very effective for highly cyclic graphs [14, 16]. In this paper, a version of this clustering technique is implemented, benchmarked on real GIS maps, and compared against alternative strategies.

Heuristic partitioning techniques [5, 7, 12, 15] commonly deployed in VLSI (Very Large Scale Integrated Circuit) design can be used for graph clustering. Such techniques are based on certain objectives, such as minimizing the total distance of inter-partition links. [14] uses the two-way parti-

tioning algorithm [5] as a clustering mechanism for the proposed access structure for aggregate queries for transportation networks. Their aggregate query experimentation considered only linear path traversal such as evaluation of an existing path. Recursive path search such as the ones discussed in this paper was not considered. In this paper, we implement the two-way partition algorithm [5] as one of the clustering optimizations for path query processing.

3 Spatial Partition Clustering (SPC)

3.1 Exploiting GIS Road Map Characteristics

The Spatial Partition Clustering (SPC) is designed to exploit the unique GIS road map characteristics to achieve I/O optimization in path query processing. It clusters link tuples in the *link table* by the spatial proximity of their *origin* nodes.

We identify the following unique characteristics of the GIS road maps managed by many digital map databases:

- GIS maps are relatively sparse, have uniform fanout typically between 2 and 5.
- GIS maps are strongly inter-connected, with each node typically reachable from near-by nodes in a few hops.
- GIS maps consist of mostly short links in comparison to their map sizes.

Because most GIS links are short, graph-traversal algorithms exhibit high expansion locality on GIS maps. Furthermore, page sizes in modern databases can be quite large, therefore many link tuples in the *link table* can be stored within one page. Since GIS maps are sparse with low fanout, multiple groups of links with the same *origin* can be stored within one page. We call them Same-Origin-link (SOL) groups. For example, with a 4KB page size and link tuple size of 128 bytes, 32 links can be stored within one page. For a GIS map with average fanout of 3, roughly 11 SOL groups can be clustered in one page. This means that there are roughly 11 different nodes in each page that could potentially be expanded by the search algorithm.

If we cluster the *link table* so that every page contains links whose *origin* nodes are closely located geographically, we are grouping the expansion nodes based on their spatial proximity. Based on the fact that GIS maps are highly interconnected and consist of mostly short links, graph-traversal algorithms such as *Dijkstra* are likely to expand nodes within the same page by traversing the intra-page links before traversing cross-page links with such a clustering. Given a fixed-sized main memory buffer not large enough to hold the entire *link table*, such paging behavior would decrease page misses caused by cross-page link traversing. We now present the algorithm that creates the spatial partition clustering for a given GIS road map.

3.2 Creating the Spatial Partition Clustering

The basic idea of SPC is first to sort all links by the x-coordinates of their *origin* nodes. A *plane-sweep* technique is then applied to sweep all x-sorted links along the x-coordinate from left to right. The sweeping process stops periodically to sort the links swept since the last stoppage by the y-coordinates of their *origin* nodes. Because the *origin* nodes of the links between two stoppage points span a short distance along the x-axis, sorting these links by the y-coordinate values of their *origin* nodes achieves a partial spatial ordering. After each y-sorting, the y-sorted links can be group into pages and written to a new *link table* that is SPC clustered.

One critical decision is to determine the proper stoppage points during plane sweeping when y-sorting takes place. Our goal is to achieve a balanced partitioning in which each resulting partition consists of links whose *origin* nodes are located within a bounding area that resembles a square block for maps with evenly distributed links. In this paper, we introduce a heuristic that dynamically computes the proper stoppage points in order to achieve such a balanced partitioning. To accommodate unevenly distributed maps, the heuristic we use will adjust the bounding block for each partition by growing in the y-axis direction if the regional link distribution is sparse, and shrinking if otherwise. In either case, each partitioned page is maximally filled with links whose *origin* nodes are relatively closely located.

To present the algorithm that creates the SPC clustering, we use the following parameters:

— f is the blocking factor for a given page size in the *link table*. We call every f consecutive link tuples an f-page.

— The *block table* is a temporary table that stores the links collected between two stoppage points during the sweeping process.

— dx_i is the difference between the minimum and maximum x-coordinate values of the *origin* nodes of the links in the first i f-pages in the *block table*.

— dy_i is the difference between the minimum and maximum y-coordinate values of *origin* nodes of the links in the first i f-pages in the *block table*.

— SPC-clustered *link table* is the resulting table.

The following is the algorithm that creates the SPC:

1 The unclustered *link table* is sorted by the x-coordinate values of the *origin* nodes of its link tuples. The result is called the x-sorted *link table*.

2 Read the x-sorted *link table* sequentially one f-page at a time, and write it to the end of the (initially empty) *block table*. Then check the following conditions:
— If all tuples in the x-sorted *link table* are read, go to step 3. (This is the end of loop.)
— If there is only one f-page in the *block table*, go to step 2 to read the next f-page and write it to the end of the *block table*.
— Otherwise, conduct the following evaluation:

2.1 Let p be the number of f-pages in the *block table*. Compute the following:

$$d_p = |(dy_p/p) - dx_p|$$

$$d_{p-1} = |(dy_{p-1}/(p-1)) - dx_{p-1}|$$

2.2 If $d_p > d_{p-1}$, this is a stoppage point. Perform the following:

2.2.1 Sort the link tuples of the first p-1 f-pages by the y-coordinate values of their *origin* nodes, group them into pages, and sequentially append to the SPC-clustered *link table*.

2.2.2 Move the p-th f-page of link tuples in the *block table* to the first page in the *block table*. Set the number of pages in the *block table* to 1.

2.3 Go to step 2 to read the next f-page.

3 Sort all the remaining link tuples in the *block table* by the y-coordinate values of their *origin* nodes, group them into pages, and sequentially append to the SPC-clustered *link table*. The SPC clustering is completed.

The intuition behind the heuristic is that when the first few f-pages (e.g., 1 or 2) are written from the *link table* to the *block table*, p is small, and $d_p = |(dy_p/p) - dx_p|$ will likely be large, assuming a map with evenly distributed links. This is because in the *plane sweep* process, we are proceeding with small progress on the x-axis and with entire range on the y-axis. When more f-pages are added to the *block table*, p increases and d_p decreases. At some point, d_p will approach 0 and then starts picking up again when $dx_p > (dy_p/p)$. We capture this point by dynamically detecting $d_p > d_{p-1}$ and then we make it a stoppage point. At a stoppage point, links in the first p-1 f-pages in the *block table* are sorted by the y-coordinate values of their *origin* nodes. Because $d_{p-1} = |(dy_{p-1}/(p-1)) - dx_{p-1}|$ approaches 0, each partition will be bounded by an area that resembles a square box.

Figure 1 illustrates the sweeping process and the heuristics in determining the stoppage points. In Figure 1(a), the link tuples are sorted by the x-coordinate values of their *origin* nodes. Next, f-pages of link tuples are written to the *block table* sequentially. In Figure 1 (b), when the 4th f-page is written to the *block table*, $d_4 > d_3$. This is a stoppage point. In Figure 1 (c), links in the first 3 f-pages in the *block table* are then sorted by the y-coordinate values of their *origin* nodes and the y-sorted links are grouped in pages and written to the SPC-clustered *link table*. Note that at this point, the first 3 f-pages in the *link table* are properly clustered. When the sweeping process is complete, all f-pages in the *link table* are properly clustered as shown in Figure 1 (d).

4 Alternative Graph Clustering Strategies

We now discuss three alternative clustering strategies implemented for comparative studies. They are the Two-Way

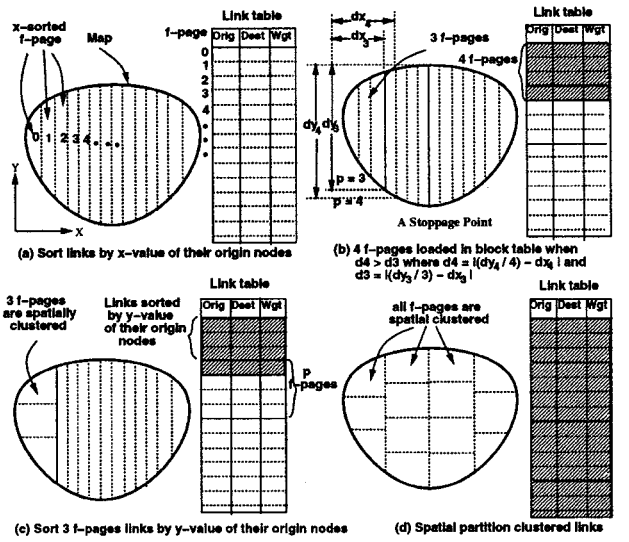

Figure 1: Spatial Partition Clustering.

Partition Clustering (TWPC) [5], the "approximately" Topological Clustering (TopoC) [2], and the Random Clustering (RandC). We assume that for each clustering technique, links of the same *origin* are always clustered together in the *link table*. Such a clustering is important because the graph-traversal path search algorithms typically expand a node by traversing all its outgoing links to the connecting nodes. Grouping links by their *origin* nodes makes sure such expansions exhibit good I/O behavior.

4.1 Two-Way Partition Clustering

Partitioning algorithms have been widely deployed in the design and fabrication of VLSI (Very Large Scale Integrated circuit) chips. Most such algorithms partition a network into two subnetworks [5, 7, 12], and through a *divide-and-conquer* process, reduce a complex problem into smaller and hence more manageable subproblems. The common objective of such partitioning is to shorten the total interconnection distance between all subnetworks in achieving a reduced layout cost and better system performance. We now propose that these partitioning algorithms could also be applied to our problem of GIS graph clustering, namely to cluster the *link table* by storing each partition within a single page. In our context, the size of each partition therefore is bounded by the size of a buffer page. Our goal of such partitioning is to reduce the page misses, to a minimum if possible, occurred during path query computation. Because each cross-page traversal in path computation may potentially incur a page miss, our partition objective is then to *minimize the number of inter-partition (cross-page) links.*

Because the partition problem with specified size constraints belongs to the class of NP-complete problems, all partition algorithms focus on finding heuristics in providing solutions in polynomial time. The most common heuristic

used in two-way partitioning is based a two-stage process [12]. First, an initial cut that separates a network into two is derived. Next, nodes are swapped between partitions for a better cut. Swapping can be done with a single node moving from one partition to the other, or with two nodes from different partitions moving to the opposite partition. During the swapping stage, priority is given to the swap that yields the best cut. Swapping can continue until it no longer creates a better cut. To avoid cyclic swapping that results in an infinitive loop, a restriction is imposed that allows one node be swapped only once during each swapping run. To remedy such a restriction, multiple iterations of swapping runs may be necessary to achieve a better result.

The two-way partitioning algorithm we implement extends the two-stage heuristics to include a contraction stage that was shown to be an improvement over the traditional two-stage approach [5]. Such an algorithm cuts a network into two partitions based on ratio-cut heuristics. To adapt it to our page clustering, we recursively apply it until each partition fits into one page. To minimize the size of the *link table*, we set the size of at least one partition to approximate that of a page during each cut. We abbreviate this clustering technique as TWPC. We now give an overview of the algorithm.

The two-way partitioning algorithm:

1 Contracting stage:

1.1 Initially, the network G has only one partition. Based on *divide-and-conquer*, recursively apply the ratio-cut routine in [15] to the partitions whose sizes are greater than a specified value p.

1.2 Based on the resulting partitions, contract G to a condensed graph G' such that each partition in G is a node in G' and each interconnection link between two partitions is a link between the two corresponding nodes in G'.

2 Swapping stage:

2.1 Randomly select a cut that creates two partitions in G'.

2.2 Iteratively apply the Fiduccia-Mattheyses algorithm [7] to the partitioned G' i times for a better swapping result, with the size constraints of the two resulting partitions set to $s1$ and $s2$. The i, $s1$, $s2$ are pre-specified parameters. The result is a min-cut of G'.

3 Restoring stage:

3.1 Restore the two partitions in G' created in step **2** by replacing each condensed node in each partition by its original nodes in the correspondent partition created in step **1**. The result is G cut into two partitions.

3.2 Apply the Fiduccia-Mattheyses algorithm on the two restored partitions in G one time, and the ending two partitions are the final result.

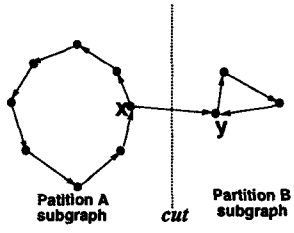

Figure 2: Partitioning with Contraction.

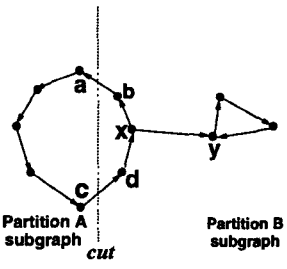

Figure 3: Partitioning without Contraction.

The ratio-cut routine [15] in step **1.1** and the Fiduccia-Mattheyses algorithm [7] in steps **2.2** and **3.3** are two min-cut algorithms based on the two-stage heuristics described previously. The intuition behind the contraction approach is that nodes that are grouped into the same partition by the ratio-cut algorithm [15] are more strongly connected. Treating them as one node reduces the chance of separating them into different partitions by a bad split. For example, in Figure 2, the ratio-cut routine may group the circle nodes and triangle nodes into two different subgraphs A and B. Because partitions A and B are subsequently contracted into two inseparable units, if there is a cut that goes through A and B, it has to go through the link between node x and node y. Note that this is an optimal cut between A and B and no further swapping will change this cut-link. If no contraction is performed, an initial cut of all the nodes in A and B may look like the cut in Figure 3. Note that subsequent swapping will not alter this cut because any single-node or pair-wise swapping of nodes a, b, c, d does not yield a cut with less inter-partition links. Therefore the optimal cut that goes through the link between nodes x and y is lost.

4.2 Approximately Topological Graph Clustering

Pure topological clustering [4, 13] arranges links in a topological order. With an adequate buffer size, such an ordering can facilitate path query processing with only one pass of the *link table*. Pure topological clustering however is not applicable to cyclic graphs such as GIS maps. [2] proposed an approach which extended topological clustering to cyclic graphs by recursively breaking cycles and preserving acyclic subgraphs for a cyclic graph. This approach is called "approximately" topological clustering.

In this paper, we implement the approximately topological clustering algorithm proposed in [2] and call it TopoC. The following is a description of the main steps of TopoC:

1 Iteratively move a root-link [5] in the graph into the clustered *link table*, until no root-link exists. If the remaining graph is empty, go to step **4**.

2 Move a sink-link [6] to a temporary *link table*. Repeat step

[5] A root-link is a link whose *origin* node has no in-coming connection.

[6] A sink-link is a link whose *destination* node has no out-going connection.

2 until no sink-link exits.

3 Randomly pick a node in the remaining graph and move all its out-going links to the temporary *link table*. Go to step **1**.

4 Append the links in the temporary *link table* in reverse order to the clustered *link table*.

In TopoC, steps 1 and 2 preserve the acyclic portions of the graph whereas step 3 breaks the cycle by removing all out-going links for a selected node. If the remaining graph at any time is acyclic, its links are topologically sorted by step 1 that continuously moves root-links to the clustered *link table*. Finally, after appending the links of the temporary *link table* in reverse order in step 4, all links are "approximately" topologically clustered in the *link table*.

4.3 Random Clustering

Random Clustering (RandC) keeps the link tuples in the *link table* in random order, except that links of the same *origin* node are clustered together. We use Random Clustering as the strawman to determine the path query processing cost when no clustering strategy is deployed.

5 Testbed Environment

Our experimental testbed is implemented on a SUN Sparc-20 workstation running the Unix operating system. It includes the clustering algorithms presented in this paper, a heap-based *Dijkstra* algorithm, an I/O buffer manager, and many other supporting data structures. All programs are written in C++.

We use *link table* to model the topology of the graph. Each *link tuple* in the *link table* models a link in the graph. The path queries discussed in this paper are assumed to be path queries with embedded constraints (see examples in Section 1). To resolve such constraints may require the retrieval of link attributes in order to evaluate the validity of each link traversed during path finding. We thus must store relevant link attributes in their respective *link tuples*. In this paper, the *link tuple* used in our experiments is set to 128 bytes. The page size is 4 Kbytes.

In our experiments, we vary the I/O buffer sizes from 64 Kbytes to 640 Kbytes and then take the average of five runs. The sizes of the *link table* of our test maps are about 2 Mbytes, with a small difference between the various clustering techniques. Note that the size of a *link table* is proportional to the size of the map. In this paper, we test maps with up to 15,000 links. For large city maps, the number of links can be many times more[7]. This means that the buffer size should also increase proportionally for large maps in order to process path queries efficiently.

We test both randomly generated graphs and a real street map of Ann Arbor City (5,596 nodes, 14,033 links). We experiment with random graphs that have 5,000 nodes and an average outdegree of 3, similar to that of the real map. We create two additional sets of random graphs, one with high locality, the other with no locality. The reason we experiment with random maps of both high and no locality is because GIS applications such as Intelligent Transportation Systems need to model graphs beyond the road transportation maps (such as the Ann Arbor city map). For example, the airline flight routes exhibit no planarity and locality therefore can be better modeled by random graphs with no locality. An intermodal map of both subway train and bus routes, however, exhibits high locality without planarity, therefore can be modeled by random graphs with high locality.

[7]For example, the Detroit map we are using for related research efforts in this project has more than 50,000 links.

6 Experimental Evaluation

This section presents experiments and performance evaluation of the various clustering techniques proposed in this paper. They are SPC (Spatial Partition Clustering), TWPC (Two-Way Partition Clustering), TopoC (Topological Clustering), and RandC (Random Clustering). Our measurement is based on a simulated number of page I/O.

We conduct single-source shortest path search for randomly selected nodes. Because the single-source shortest path computation for a source node i corresponds to the search of the longest shortest-path from node i for graph-traversal algorithms, this set of experiments tests the worst-case scenario in searching a shortest path from node i.

6.1 Experiments on the Real Map

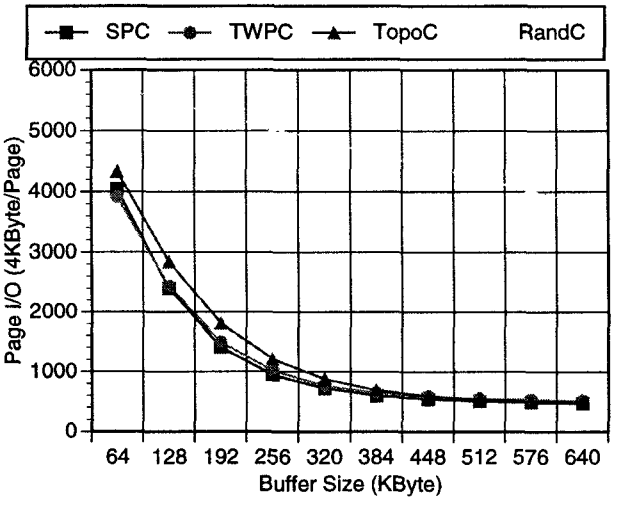

Figure 4: I/O Cost on Real Map.

In the first set of experiments, we use the real Ann Arbor map. The results in Figure 4 show that Random clustering performs much worse than any other clustering, confirming our claim in this paper that proper graph clustering can be a key to efficient path query processing. Because the cost of RandC is very high, making it hard to see the difference in performance between the other three clustering approaches, we

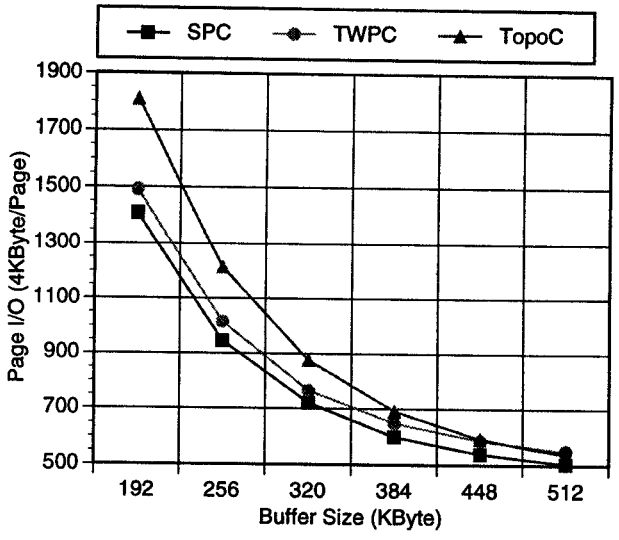

Figure 5: I/O Cost on Real Map (excl. RandC).

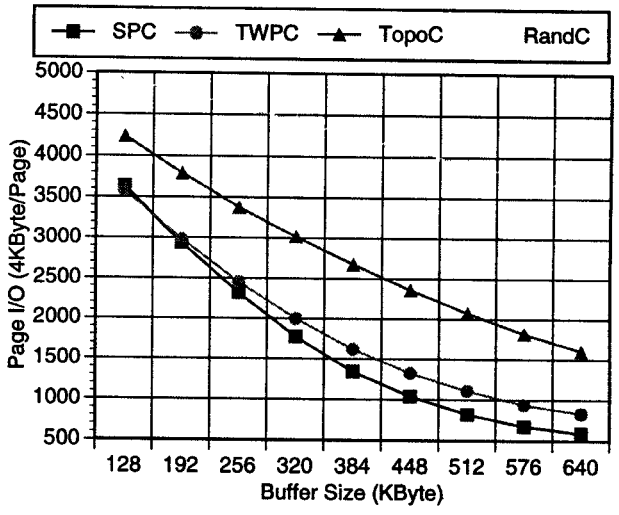

Figure 6: I/O Cost on High-locality Graph.

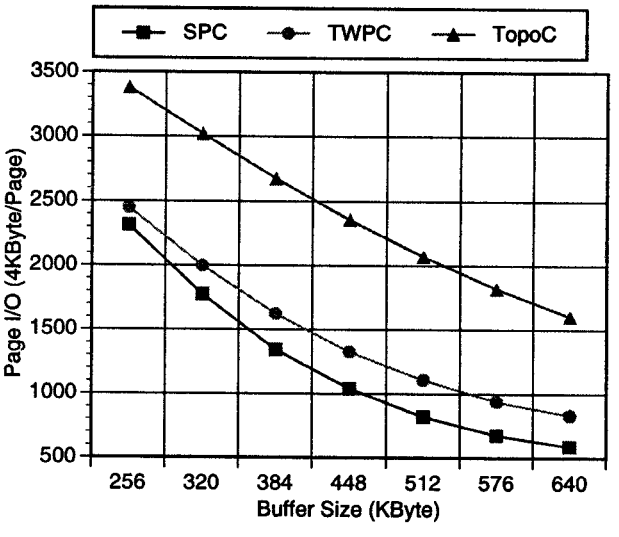

Figure 7: I/O Cost on High-locality Graph (excl. RandC).

plotted Figure 5 without showing the RandC results. Figure 5 shows that SPC performs the best, followed by TWPC and then TopoC. The results indicate that for high-locality graphs such as the Ann Arbor map minimizing the cross-page links is not as effective as partitioning based on spatial proximity. Although TopoC has the worst performance among the three clustering optimizations, it is still much more effective than RandC. This is contradictory to the suggestion in [14, 16] that topological clustering is not effective for highly cyclic graphs. Note that when the buffer size is greater than 512 Kbytes, the difference between the three clustering strategies is becoming small. This is because the size of the buffer is large enough to contain the expansion locality of the *Dijkstra* algorithm captured by all three clustering optimizations. Therefore, roughly one pass for such a large buffer would be sufficient to compute the single-source shortest paths.

6.2 Experiments on the High-Locality Random Graphs

The second set of experiments is based on a randomly generated graph with 5,000 nodes, average outdegree of 3, and high locality. While the real Ann Arbor map is very planar and interconnected, the high-locality random graph does not guarantee planarity. We conduct the same single-source path search experiments described above. The results in Figure 6 show that RandC remains the distant worst, with the TopoC significantly worse than the other two clustering approaches. The close-up results in Figure 7 show that SPC still performs better than TWPC. This means that high locality is the dominant reason why SPC performs better than TWPC. Planarity is less relevant.

6.3 Experiments on the Low-Locality Random Graphs

Lastly, we test a randomly generated graph with 5,000 nodes, average outdegree of 3, and no locality. Interestingly, the results in Figure 8 show that RandC and SPC are equally the worst. This can be explained by the fact that without locality, the spatial proximity is irrelevant in clustering optimization. Consequently, the SPC performs the same as the RandC on graphs with no locality. The results show that TWPC has the best performance, followed by TopoC. This indicates that TWPC is not locality-dependent, therefore has better performance than SPC, TopoC, and RandC on graphs with no locality.

7 Conclusions

In this paper, we consider the optimization of path query processing based on graph clustering techniques for digital map databases. Clustering optimization is attractive because it does not incur any run-time cost, and it requires no auxiliary data structures that demand memory. More importantly, it is complimentary to many of the existing path query solutions proposed in the literature that typically tackle the problem at the data structure or at the algorithmic level.

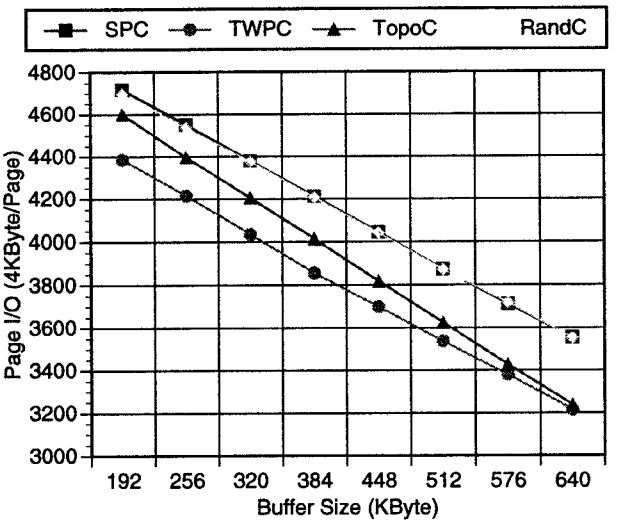

Figure 8: I/O Cost on Low-locality Graph.

The contributions of this paper are:

1 A new clustering technique, called spatial partition clustering, is developed for path query optimization for digital map databases.

2 Three existing graph clustering techniques, namely two-way partitioning, topological, and random clustering, are identified. All four techniques are implemented in a uniform testbed for a fair benchmarking of path query processing in digital map databases.

3 For the first time, experimental evaluation of the comparative performance of the above four graph clustering techniques is conducted based on various buffer sizes and different graph localities.

4 Our experimental results can be used to establish guidelines in selection of the best clustering technique based on the type of maps at hand. For example, the results show that our spatial partition clustering performs the best for high-locality graphs whereas the two-way partitioning approach works the best for low-locality graphs.

References

[1] 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.

[2] Agrawal, R. and Kiernan, J., "An Access Structure for Generalized Transitive Closure Queries", *IEEE 9th Int. Conf. on Data Engineering*, 1993, pp. 429 – 438.

[3] Bancilhon, F. and Ramakrishnan, R., "An Amateur's Introduction to Recursive Query Processing Strategies," *Proc. of the 1986 ACM SIGMOD Int. Conf. on Management of Data*, 1986.

[4] Banerjee, J., Kim, W., Kim, S.J., and Garza, J.F., "Clustering a DAG for CAD Databases," *IEEE Trans. on Software Engineering*, Vol. 14, No. 11, 1988.

[5] Cheng, C.K. and Wei, T.C., "An Improved Two-Way Partitioning Algorithm with Stable Performance," *IEEE Trans. on Computer-Aided Design*, Vol. 10, No. 12, Dec. 1991, pp. 1502 – 1511.

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

[7] Fiduccia, C.M. and Mattheyses, R.M., "A Linear Time Heuristic for Improving Network Partitions", *Proc. ACM/IEEE 19th Design Automat. Conf.*, 1982, pp. 175 – 181.

[8] Huang, Y. W., Jing, N. and Rundensteiner, E., "Hierarchical Path Views: A Model Based on Fragmentation and Transportation Road Types," *The Third ACM Workshop on Geographic Information Systems*, Washington, D.C., Nov. 1995, pp. 93 – 100.

[9] Huang, Y. W., Jing, N. and Rundensteiner, E., "Evaluation of Hierarchical Path Finding Techniques for ITS Route Guidance," *Proc. of ITS-america*, Houston, April, 1996.

[10] 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.

[11] Jing, N., Huang, Y.W., and Rundensteiner, E., "Hierarchical Optimization of Optimal Path Finding for Transportation Applications," *Proc. of the 5th Int. Conf. on Information and Knowledge Management*, 1996.

[12] Kernighan, B.W. and Lin, S., "An Efficient Heuristic Procedure for Partitioning Graphs," *Bell Syst. Tech. J., vol. 49, no. 2* Feb. 1970, pp. 291 – 307.

[13] Larson, P.A. and Deshpande, V., "A File Structure Supporting Traversal Recursion," *Proc. of the 1989 ACM SIGMOD Int. Conf. on Management of Data*, May 1989, pp. 243 – 252.

[14] 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.

[15] Wei, Y.-C. and Cheng, C.-K, "Ratio Cut Partitioning for Hierarchical Designs," *Tech. Rep. CS90-164, Univ. California, San Diego*, Jan. 1990.

[16] Zhao, J.L. and Zaki, A., "Spatial Data Traversal in Road Map Databases: A Graph Indexing Approach," *Proc. of the 3th Int. Conf. on Information and Knowledge Management*, 1994, pp. 355 – 362.