# Optimizing Path Query Performance: Graph Clustering Strategies *

**Yun-Wu Huang**[†]
IBM T.J. Watson Labs
ywh@us.ibm.com

**Ning Jing**[‡]
Changsha Institute of Technology
jning@eecs.umich.edu

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

## Abstract

Path queries over transportation networks are operations required by many Geographic Information Systems applications. Such networks, typically modeled as graphs composed of nodes and links and represented as link relations, can be very large and hence often need to be stored on secondary storage devices. Path query computation over such large persistent networks amounts to high I/O costs due to having to repeatedly bring in links from the link relation from secondary storage into the main memory buffer for processing. This paper is the first to present a comparative experimental evaluation of alternative graph clustering solutions in order to show their effectiveness in path query processing over transportation networks. Clustering optimization is attractive because it does not incur any run-time cost, requires no auxiliary data structures, and is complimentary to many of the existing solutions on path query processing. In this paper, we develop a novel clustering technique, called spatial partition clustering (SPC), that exploits unique properties of transportation networks such as spatial coordinates and high locality. We identify other promising candidates for clustering optimizations from the literature, such as two-way partitioning and approximate topological clustering. We fine-tune them to optimize their I/O behavior for path query processing. Our experimental evaluation of the performance of these graph clustering techniques using an actual city road network as well as randomly generated graphs considers variations in parameters such as memory buffer size, length of the paths, locality, and out-degree. Our experimental results are the foundation for establishing guidelines to select the best clustering technique based on the type of networks. We find that our SPC performs the best for the highly interconnected city map; the hybrid approach for random graphs with high locality; and the two-way partitioning based on link weights for random graphs with no locality.

*Index Terms* — Path Query Processing, Transportation Networks, Spatial Clustering, Clustering Optimization, Geographic Information Systems.

* 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. N. Jing was supported in part by the State Education Commission of P.R. China .

[†] This work was performed while the author was Ph.d. student at the University of Michigan

[‡] 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.

# 1 Introduction

## 1.1 Background on Path Query Processing

Transportation networks are essential components of many Geographic Information Systems (GIS) applications. Such applications include navigation, route guidance, traveler information systems, fleet management, public transit, troop movement, urban planning, to name a few. Among the services provided by such GIS systems, path query processing is an important feature required by many of the above applications [17, 22, 23, 25, 24, 37, 40]. 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 for more than 10 miles.*"

In addition to requiring path search support from the GIS system, queries such as the three above often also have embedded constraints that must be processed. For example, for **Q1**, the computed path contains no links of type *toll* road. This alpha-numeric filter can be applied to all links of the graph before path processing, thus effectively constructing a smaller subgraph on which to apply the path search. For **Q2**, the constraint is that the destination nodes are of *garage* type. Again prefiltering could be conducted. For **Q3**, we now deal with a spatial and hence much more expensive filter, namely that the computed path does not contain any link that traverses areas with *altitude* $>$ *1000 feet for more than 10 miles.* In this case, we cannot a priori determine a valid subgraph of the complete network and worse yet, the spatial characteristics have to be stored with each link – thus significantly increasing the storage requirements and thus expected costs of path processing. Note that while the cost measurements used in path query computation for all examples may be different, such as based on fuel consumption in **Q1**, on travel time in **Q2**, and on distance in **Q3**, they can be all abstracted and then handled using the same path search techniques.

In order to process path queries such as the above, a GIS system must model the topological information of the transportation networks as well as maintain the attributes and cost measurements associated with each component of the network. Typically, a GIS system models the topological information of a transportation network by representing it as a graph composed of nodes and links.

A node represents for example 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. The network then stores the topological information and other *attributes* associated with intersections and road segments in two separate structures, called the *node table* and the *link table* respectively in this paper. Each element in such a table is referred to as a tuple of the table. The attributes that describe a node tuple may include its x- and y-coordinates, the connecting road segments (incoming and outgoing), the traffic control configuration (traffic light, stop sign, etc.), points of interest, and so on. A link itself is identified by its *origin* and *destination* nodes. Additional attributes for describing each link include for example for a road network the number of lanes, maximum speed, length, up-to-date link travel speed, and so on. The sizes for each node and link therefore can be very large, up to hundreds of bytes in length.

Transportation networks are considered stable graphs for the purpose of this paper, since the addition and the removal of intersections or roads occurs only very infrequently in practice. The cost measurement data used for path query computation may however be either stable or unstable depending on the attributes. The up-to-date estimated link traversal time, for example, may depend on changing traffic conditions, and therefore is unstable because it needs to be updated as soon as the traffic changes occur. Link distance or the geographic coordinates of nodes on the network on the other hand are considered stable.

## 1.2 Clustering for Path Query Processing: Motivation

This paper investigates the optimization of path query processing based on graph clustering techniques. To compute paths for path queries such as those previously listed (**Q1 - Q3**), we assume that popular graph-traversal search algorithms such as the *Dijkstra*, $A^*$, Breadth-First Search, and Depth-First Search algorithms or any of their variants are used. They search for paths by traversing from one node to another through their respective connecting link. Because path search computation is recursive in nature, searching a path means to recursively access links from the *link table*.

However, since the size of the *link table* is often larger than the capacity of the main memory buffer of a given GIS system, the *link table* may need to be stored on a secondary storage device, typically on disk. While state-of-the-art database engines may attempt to cache the link table into main memory during path evaluation, this will generally not be feasible due to size constraints.

In this case, many tuples (links) in the *link table* may need to be retrieved over and over again from secondary storage and placed into the main memory buffer for evaluation. Given that such I/O operations on most modern computers are typically several 100-fold more expensive than CPU operations, the I/O costs are the dominant factor of path computation costs.

The high processing costs are thus incurred by the recursive nature of the graph traversal component of path query computation. Resolving embedded constraints may further increase I/O costs significantly. For example, in a related effort [17, 18, 22, 23, 19, 20], we found that processing spatial constraints (see **Q3** path query) is very I/O intensive. Thus such constraint resolution competes with the path finding component of the search process for computational resources such as the buffer space. This further motivates our research presented in this paper on optimizing the path computation process by reducing I/O activities.

Data is commonly not transferred between secondary storage and main memory one tuple at a time, but rather at the granularity of one or more buffer pages containing possibly many tuples each. Hence, one important performance consideration studied by the database community is how best to place tuples onto disk pages so to minimize the number of required I/O operations. This is done by assuring that tuples brought into memory on one disk page are ideally all made use off whenever in the buffer. This optimization strategy of grouping data onto pages is commonly referred to as *clustering.*

The purpose of this paper is to demonstrate that clustering optimization for path query computation can be effective for many types of transportation networks. Clustering is attractive because it does not incur any run-time cost, nor does it require any auxiliary data structure that demands buffer space. Because transportation networks are stable graphs, clustering is a one-time apriori cost not affecting actual path processing. If the clustering were not based on not stable attributes (see Section 1.1), it may need to be reclustered at some point which again could be done off-line without any dynamic update cost . Most importantly, clustering is at a level lower than many other path query solutions that focus on auxiliary access structures or on algorithmic techniques, therefore results emerging from the comparative evaluation of our clustering research can be deployed by such other solutions that do not already employ specific link clustering [2, 3, 8, 40]. Our work thus is complimentary to much of the existing work on path finding and could be exploited to further optimize such techniques.

## 1.3 Contributions

The contributions of this paper can be summarized as follows:

**(1)** We identify existing clustering techniques from the literature, namely topological clustering (adopted from [5]), two-way partition clustering (adopted from [11]), and random clustering, and apply them for optimizing the I/O behavior for path query processing.

**(2)** We develop a novel clustering technique, which we call Spatial Partition Clustering (SPC), that exploits unique properties of transportation networks such as spatial coordinates or high locality of such networks. The basic idea of SPC is to achieve spatial partitioning by dividing the links in the networks into partitions such that the *origin* nodes of all links within a partition are bounded by an area that resembles a square on the map underlying the network. the network.

**(3)** In addition, we develop two extensions to the above techniques further tuned for path query processing for GIS systems. Namely, we combine our proposed spatial partition solution with the min-cut technique to create a hybrid approach. Second, we modify the two-way partition clustering to consider both connectivity and link weights in determining partitions.

**(4)** We implement the six selected clustering techniques on a uniform testbed for fair comparison. To benchmark these six techniques, we also implemented the *Dijkstra* algorithm [12], one of the more popular and effective path search algorithms, so that it can be applied as search technique over the network data once clustered.

**(5)** We perform extensive experimental evaluation of both existing as well as our proposed clustering techniques to determine their relative effectiveness. These experiments are conducted both using real data (in particular, the road network of Ann Arbor, Michigan (5,596 nodes, 14,033 links)) as well as synthetically generated networks. We compare the performance results of running the *Dijkstra* algorithm on network data layed out on disk by the different clustering strategies.

**(6)** Our experimental results are used to establish guidelines to select the best clustering technique based on the type of network found in a given application domain. Characteristics of the network considered include parameters such as the size of the graph, the average out-degree

or the locality[1].

While a preliminary version of this work appears in an earlier conference paper [26], this journal paper differs from it in many respects. First, we propose two additional new graph clustering techniques that are the extensions of the clustering techniques presented in [26], namely the hybrid SPC technique and the link weight based partitioning technique. Second, we include the experimental results of the new extensions into the performance evaluation section in this paper. Third, this paper presents additional types of experiments that provide new insights into the behavior of proposed clustering techniques. Such experiments, not available in the previous report, are for example the path finding experiments based on paths of different lengths and networks of various average out-degrees. These new types of experiments have been instrumented for all clustering techniques, both the ones introduced in [26] as well as the new optimizations. Fourth, because the performance of the new extensions can be shown to lead to further improvement over the original techniques, the conclusions for this paper have been revised to reflect the new results. Lastly, this paper is written with more examples and illustrations for better understanding and accessibility to the material.

## 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 fixed-size main memory I/O buffer. Most of such research has proposed solutions to solve recursive query problems for *general databases* that focused on pure transitive closure computation [1, 4, 7, 13, 27, 28, 29, 35]. In our work, rather than aiming for generality, we now take an application-driven stance by proposing different disk page clustering algorithms for optimizing path query processing for GIS type of applications and then experimentally evaluating their relative advantages and disadvantages.

Two unresolved problems arise when applying transitive closure pre-computation techniques to path query processing for transportation networks. First, a single transitive closure computation cannot take different embedded constraints into account. For example, a transitive closure computed for path query **Q1** cannot be used to answer path query **Q3**. To answer all path queries with

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

a large set of different embedded constraints, we may need to compute numerous transitive closures, each based on a unique embedded constraint. Clearly, this is not feasible in practice. Second, some link weights (cost measurements) may be unstable and can change very frequently. In order for the transitive closure computed based on such cost measurements to reflect the most up-to-date cost, re-computation may need to be conducted very frequently. However, performance results in [1, 29] have shown that their techniques are not efficient in computing the shortest path transitive closure for graphs with cycles such as transportation networks in GIS applications. Re-computation of shortest path transitive closure using such techniques thus cannot be done frequently, under-cutting the correctness of the computed paths.

In the GIS community, there has been much work on data structures and representations for most efficiently being able to manage and access geograhical data sets [15, 16]. Focus here typically was on the logical organization of data to support certain access patterns, whereas less effect has gone into the lower-level physical aspect of management of the data on disk (clustering). The later is the target of our current work, and is complimentary to data structure type of work. Similarly, there is also a body of literature on evaluating shortest path algorithms both in the database as well as in the GIS community. Focus here has been on a comparison of the performance of different types of shortest path algorithms itself [39], on on-line reordering [24], or on a-priori materialization of best paths [21].

In [2], a multi-level structure is proposed to prune the search space by conducting path pre-computations within each component data structure. Such a technique is less useful for processing path queries with embedded constraints because the pre-computation information is general and may not be effectively usable as heuristics in pruning the search space for path queries with embedded constraints.

In our previous work, we have explored the hierarchical path view approach which fragments a large graph into smaller subgraphs and pre-computes the path transitive closure for each subgraph [30, 23, 25, 31, 21]. The advantage of such a technique is more efficient computation in both transitive closures of the subgraphs and path search through the hierarchy.

In [40], a graph indexing technique is proposed to improve paging performance for graph traversal by building an auxiliary structure that predicts which nodes are to be accessed in the future. However, such an auxiliary structure itself requires storage that competes for buffer space. A further

limiting factor is that the examples given in [40] have fanout of only 1 or 2. Typical transportation networks have a fanout of at least 2 and 5, meaning that the size of the auxiliary structure will be larger than indicated in [40]. Thus more buffer space needs to be put aside to load such a structure. In [40], no path finding experiment on real networks was conducted. The degree of improvement for GIS road and other transportation networks therefore is unknown.

Clustering techniques, which are the focus of this paper, have been previously proposed to reduce the path query processing I/O costs by arranging the link relation in a certain order in secondary storage [5, 6, 32]. However, pure topological clustering [6, 32] is not effective for shortest path computation for cyclic graphs such as GIS transportation networks where the ancestor relation is mostly bidirectional. This is because pure topological clustering only preserves the ancestry relation in one direction; it does not guarantee good paging behavior when the computation searches the other direction when cycles exist. In [5], an approximate topological clustering was proposed that handles cyclic graphs using heuristics that first remove the acyclic subparts of the graph and next remove some links to break the remaining cyclic subparts. The goal of such clustering is to minimize the number of link tuples that trace backward topologically. In this paper, a version of this clustering technique is implemented and benchmarked in order to apply it to networks and to compare against alternative strategies. Our experimental evaluation in Section 6 indicates that several alternative clustering strategies outperform such topological clustering technique for both GIS networks and for random ones.

The heuristic partitioning techniques [11, 14, 33, 38] commonly deployed in VLSI (Very Large Scale Integrated Circuit) design can also be adopted to cluster the link relation for a graph. Such techniques partition a graph in subgraphs based on certain partitioning objectives, of which the most common one is to minimize the total distance of inter-connection links between partitions. [37] adopted a two-way partitioning algorithm [11] as a clustering mechanism for the proposed access structure for aggregate queries for transportation networks and found it effective. Their aggregate query experimentation given in [37] however is based on a linear path evaluation. Recursive path search such as the path queries discussed in this paper is not considered. In this paper, we also include the two-way partition algorithm [11] in our evaluation in order to compare this technique to alternative clustering solutions.

# 3 Spatial Partition Clustering

In this section, we first present the transportation network characteristics exploited by our proposed Spatial Partition Clustering (SPC) algorithm, followed by a description of the algorithm that creates the SPC.

## 3.1 Exploiting Characteristics of Transportation Networks

We propose to cluster link tuples from the *link table* based on the spatial proximity of their *origin* nodes, that is, to group tuples of the link table into disk pages and then to transfer the link relation between secondary storage and main memory in the granularity of these pages. We call this the Spatial Partition Clustering, or short SPC. To understand why the SPC can be effective for transportation networks, we describe the unique characteristics of the (road) networks:

- road networks are relatively sparse, have uniform fanout typically between 2 and 5.
- road networks are strongly inter-connected, with each node typically reachable from near-by nodes by traversing only a few links.
- road networks consist of mostly short links in comparison to the size of the underlying spatial region. In other words, most road links span a short distance from one intersection to the neighboring intersection.

Graph-traversal search algorithms conduct node expansions by traversing links. Because most road links are short, these algorithms therefore exhibit high expansion locality on transportation networks. Furthermore, page sizes in modern databases can be quite large. Therefore many link tuples in the *link table* can be stored within one page. Because road transportation networks are sparse with low fanout, multiple groups of links with the same *origin* can be stored within one page. We call them SOL (Same-Origin-link) groups. For example, with a 4 K-byte page size and link tuple size of 128 bytes, 32 links can be stored within one page. For a transportation network 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 geographically

closely located, we are grouping the expansion nodes based on their spatial proximity. Based on the fact that transportation networks are highly inter-connected and consist of mostly short links, the 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 reduce page misses caused by cross-page link traversing. We now present the algorithm that creates the spatial partition clustering for a given network.

### 3.2 The Spatial Partition Clustering Algorithm

The algorithm that creates the SPC clustering is based on the *plane-sweep* techniques commonly found in multi-dimensional spatial data operations. The *plane-sweep* technique is for example used to implement the spatial intersect operation in [9, 34, 36]. The basic idea of SPC is first to sort all links by the x-coordinate values of their *origin* nodes. The *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 last stoppage by the y-coordinate values 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 grouped into disk pages. We call the output of this clustering process the SPC-clustered *link table*, which as explained earlier corresponds to the layout of link tuples onto disk pages.

One critical decision to such a partition algorithm 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 when the links are evenly distributed on the map. Below, 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 adjusts 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:

— $f$ refers to the number of link tuples that fit into a given page size, referred to as the link tuple blocking factor. We thus 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.

**Algorithm:** SPC()

**Input:** L: link table filled with all link tuples

**Output:** CL: link table clustered into pages

**1** The unclustered *link table* L 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 using the following process: read the next $f$-page and write the page to the end of the *block table* (*block table* is initially empty). Then check the following conditions:

— If all tuples in the x-sorted *link table* are read, go to step 3.

— 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 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* CL. Output CL.

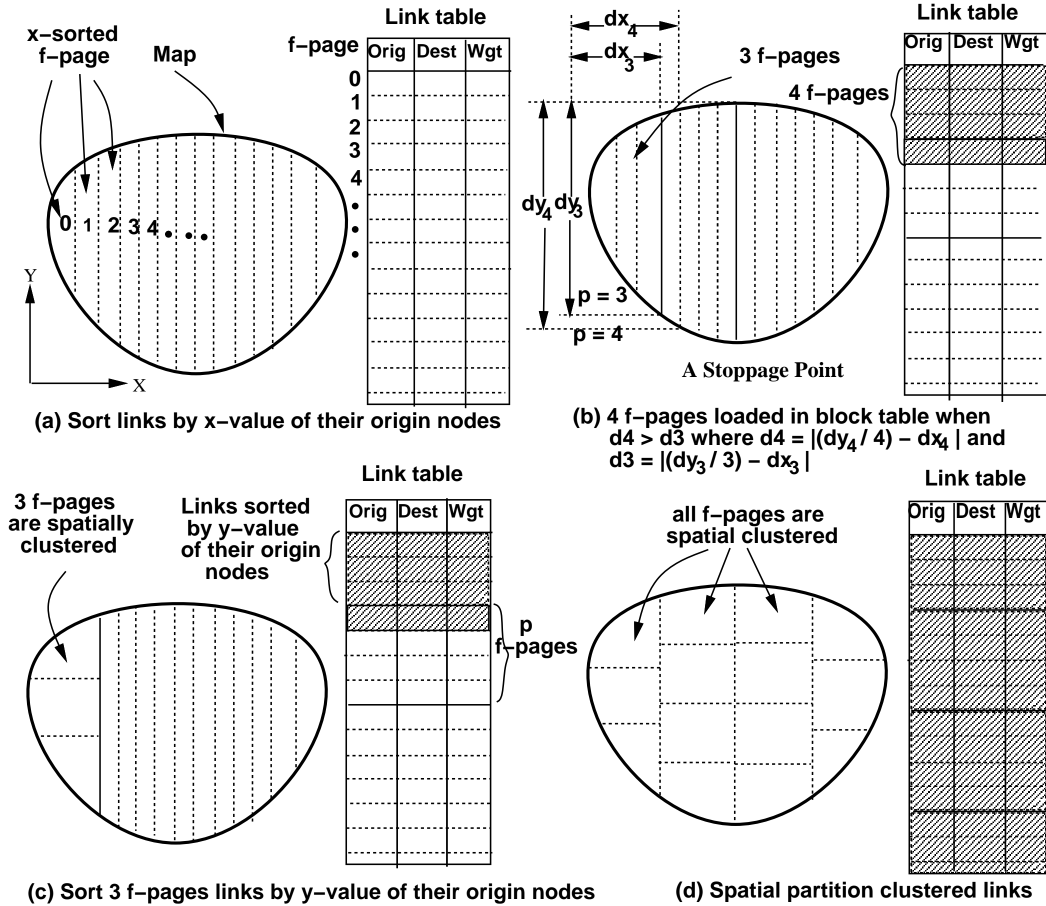


Figure 1: Spatial Partition Clustering.

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 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 reasoning behind 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

In this section, we present three alternative clustering strategies implemented for comparative studies. They are the Two-Way Partition Clustering (TWPC) [11], the "approximately" Topological Clustering (TopoC) [5], and the Random Clustering (RandC). We assume that for each clustering technique, links of the same SOL group 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 [11, 14, 33], 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 transportation network 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 that occur during path query computation to a minimum. 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.*

### 4.1.1 The Two-Way Partitioning Algorithm

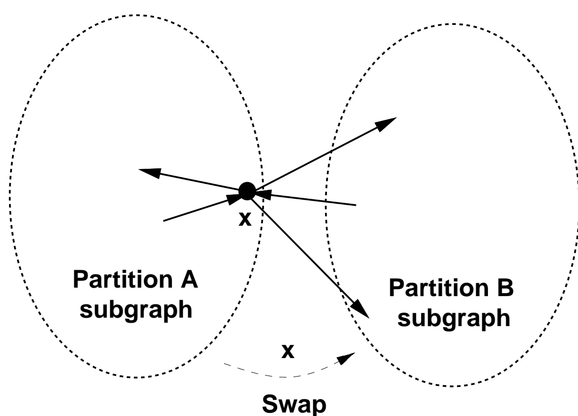


Figure 2: An Example of Single-Node Swapping.

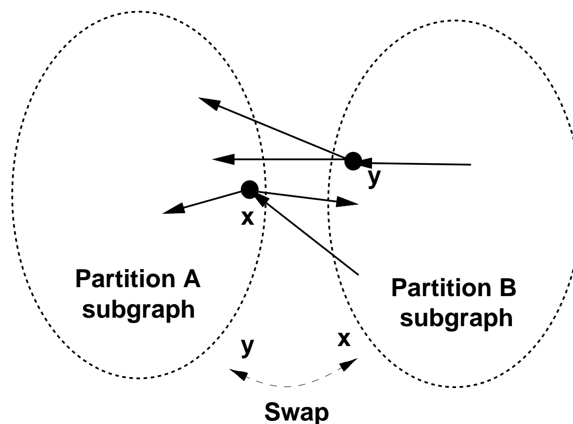


Figure 3: An Example of Pair-Wise Swapping.

We implement a partition clustering based on the two-way algorithm proposed in [11]. Because the partition problem with specified size constraints belongs to the class of NP-complete problems [10], all partition algorithms focus on finding heuristics in providing solutions in polynomial time. The most common heuristic used in two-way partitioning is based on a two-stage process [33]. First, an initial cut that separates a network into two is derived. Next, nodes are swapped from one partition to another as long as such swapping results in a better cut. For example, in Figure 2, swapping node $x$ from partition A to B creates a cut that reduces the connecting links from 3 to 2 between the two partitions. Swapping can also be conducted between two nodes from different partitions. For example, in Figure 3, swapping node $x$ and node $y$ to their opposite partition reduces the cross-partition links from 4 to 2. During each swapping run, priority is always 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 to be swapped only once during each swapping run. To remedy such a restriction, multiple iterations of swapping runs may be necessary to achieve an acceptable result.

The two-way partitioning algorithm we implement is also based on the two-stage heuristics [11]. We add a contraction stage that has been shown to be an improvement over the traditional two-stage approach [11]. Because our partitioning objective is to reduce the inter-connection links, we abbreviate this clustering technique as TWPC_con. We now give an overview of the algorithm, while a detailed presentation of the algorithm can be found in the original paper [11].

**Algorithm: Two-Way Partitioning TWPC_con().**

**Input:**

. G: network with all link tuples

. p: integer denoting maximal size of a partition

. i: integer denoting number of swaps allowed

. s1,s2: integers denoting size constraints on partitions

**Output:**

. G': network G partitioned into two groups

**1 Contracting stage:**

**1.1** Initially, the network $G$ has only one partition.

**1.2** Based on *divide-and-conquer*, recursively apply the ratio-cut routine in [38] to the partitions whose sizes are greater than a specified value $p$.

**1.3** 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 partitions $G'$. into two groups.

**2.2** Iteratively, apply the Fiduccia-Mattheyses algorithm [14] to the partitioned $G'$ $i$ times for better swapping result, with the size constraints of the two resulting partitions set to $s1$ and $s2$. The $i$, $s1$, $s2$ are pre-specified input parameters.

**2.3** The result of step 2.2 is a two-way 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 two-way cut in $G$.

**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.

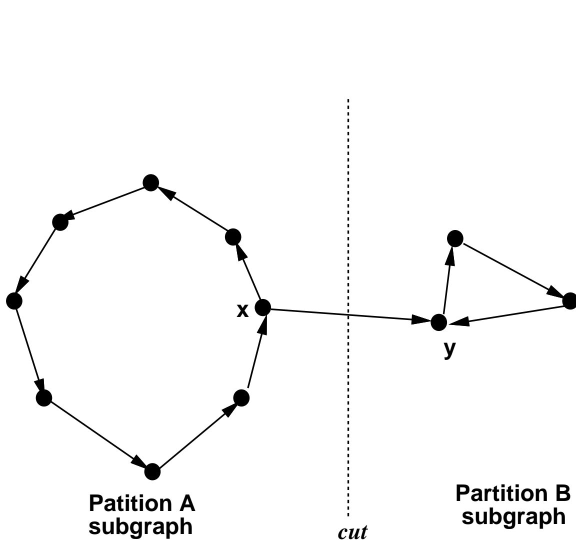


Figure 4: An Example of Partitioning with Contraction.

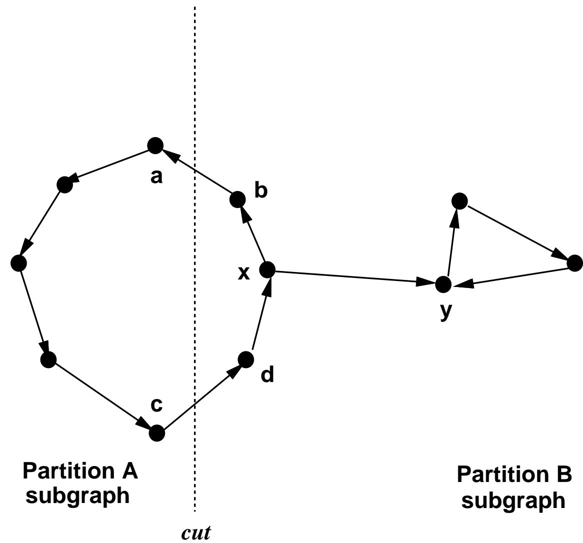


Figure 5: An Example of Partitioning without Contraction.

The ratio-cut routine [38] in step 1.2 and the Fiduccia-Mattheyses algorithm [14] in step 2.2 are two partitioning algorithms based on the two-stage heuristics described earlier. The former relies on the ratio-cut property to achieve a more balanced split while the latter deploys a data structure that reduces the computational complexity and a specific size restriction to achieve a desired partitioning.

The intuition behind the contraction approach is that nodes that are more strongly connected are identified by the ratio-cut routine in step 1.2 and treated as one node in the swapping stage. This way the chance of them being split into different partitions by a bad split is reduced. For example, in Figure 4, the ratio-cut routine may group the nodes forming a circular versus a triangular configuration 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 5. 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 $y$ would be lost in the case if we were to not utilize the contraction heuristic.

### 4.1.2 Our Adaptation of the Two-Way Partitioning Algorithm

The above two-way partition 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. Because the tremendous potential I/O required to process a path query, we desire to load as much graph information into one page as possible. Consequently, the occupancy rate of each page is set to be very high. As a result, we allow for an uneven two-way partitioning as long as the size of one partition approximates that of a page. To achieve this, we set a relatively high minimum occupancy rate for each partition. To avoid a "local minima" trap, we allow the swapping process to overflow a partition to make it larger than a page with the hope that further swapping will decrease its size to be within a page. Thus it is possible that after several iterations of swapping, one partition may have a size that is slightly greater than that of a page. To avoid partitions with such an unsatisfactory occupancy rate in our context of paging, we instead introduce a heuristic during the final Fiduccia and Mattheyses algorithm run in step 3.2 to favor swapping nodes out of such partitions whose sizes are slightly over that of a page. We find that this adaptation achieves a uniformly high occupancy rate among all final partitions.

## 4.2 Approximately Topological Graph Clustering

For an acyclic directed graph, topological clustering consists of arranging all its links in a topological order. The advantage of this topological clustering of links is that a path query over it can be processed by accessing links in one pass. Thus, if the link table does fit into main memory, the path query will exhibit good I/O performance by avoiding to access any page more than once. However, this topological clustering approach is not applicable to cyclic graphs as there is no topological order that can be established due to cycles. [5] proposed an approach which extended the topological clustering to cyclic graphs by recursively breaking cycles and removing acyclic portions of the cyclic graph. By this approach, the cyclic graph can be "approximately" topologically sorted.

In this paper, we implement and evaluate the approximately topological clustering algorithm proposed in [5] and call it TopoC (for Topological Clustering). The following is a description of the main steps of the TopoC.

**Algorithm:** TopoC()
**Input:** L: link table representing transportation network
**Output:** CL: link table clustered into pages

**1** Move a root-link [2] of the link table L into the clustered link table CL. Repeat this process until no more root-links can be found in the remaining table If the remaining link table is empty, go to step **4**.

**2** Move a sink-link [3] to a temporary link table. Repeat this process until no more sink-links remain in the link table L.

**3** Randomly pick a node in the remaining link table L. Move all its outgoing 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 CL.

The TopoC achieves approximately topological clustering in three phases: Steps 1 and 2 remove the acyclic portions of the graph. Step 3 breaks cycles by removing all out-going links of one selected node. When the remaining graph is empty, step 4 appends the links in the temporary link table to the resulting clustered link table.

As an example, Figure 6(a) shows a cyclic directed graph and its unclustered link table. We use this graph to illustrate how the clustered link table is built step by step by the TopoC algorithm. Figure 6(b) depicts the graph after TopoC moves root-links $L_{fc}$ and $L_{fg}$ into the clustered link table. Since Figure 6(b) has no sink-links, TopoC skips step 2. In Figure 6(c), TopoC breaks a cycle by moving link $L_{ab}$ into the temporary link table. Repeating step 1, Figure 6(d) depicts the graph by moving root-links $L_{bd}$ and $L_{be}$ to the clustered link table. Then step 2 removes sink-link $L_{ca}$ from the graph into the temporary link table (Figure 6(e)). By breaking another cycle in Figure 6(f), link $L_{de}$ is moved over to the temporary link table. Since the remaining graph at this time is an acyclic graph, its links are topologically sorted by repeating step 1 to move root-links to the

[2]If the *origin* node of a link has no incoming link, this link is referred to as root-link.
[3]If the *destination* node of a link has no out-going link, and the *origin* node of this link does not belong to any of the cycles, this link is referred to as sink-link.

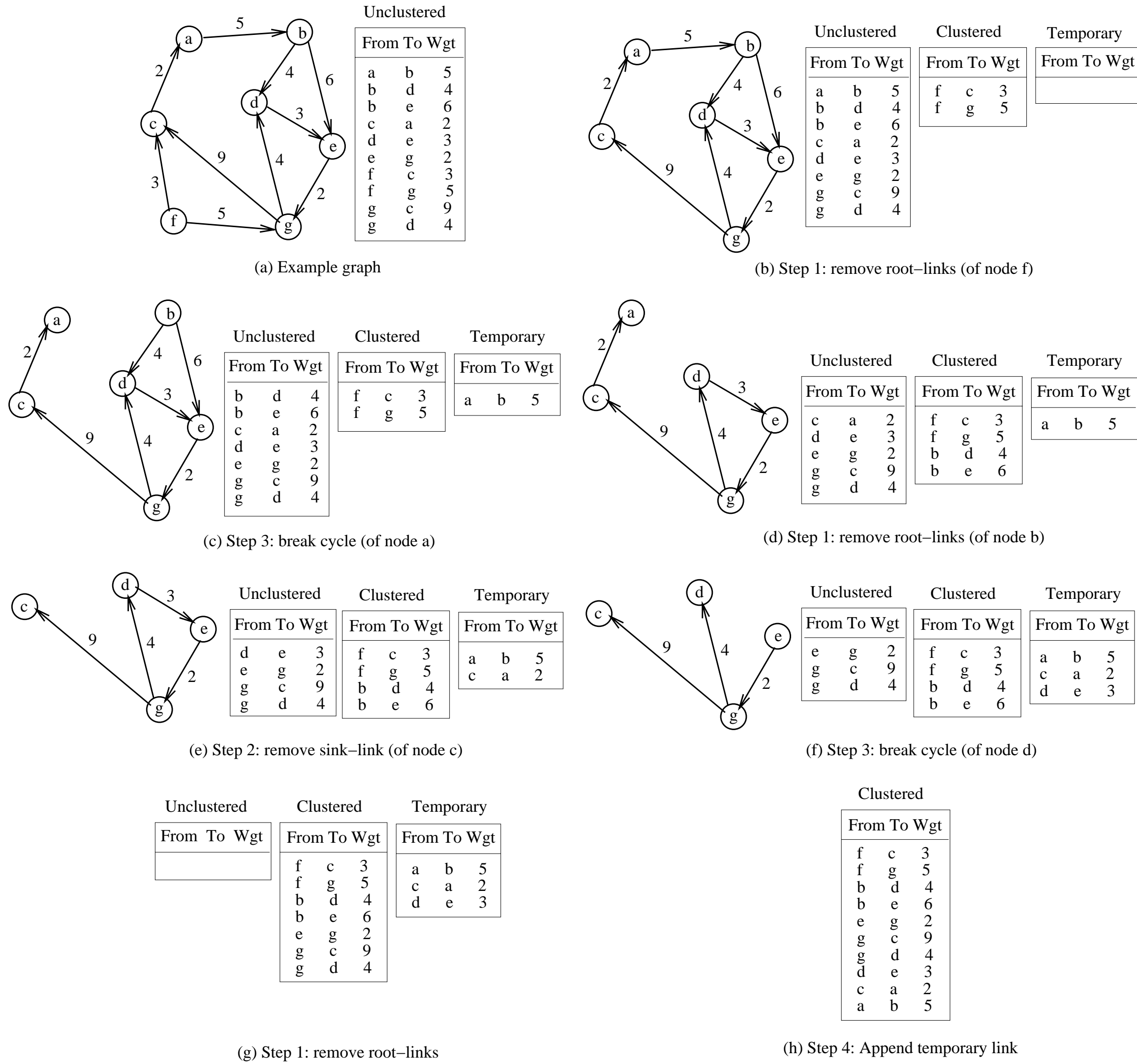


Figure 6: Example of Approximately Topological Clustering (TopoC) Algorithm.

clustered link table (Figure 6(g)). Finally, Figure 6(h) depicts the topologically clustered link table by appending the links of the temporary link table in reverse order.

## 4.3 Random Clustering

Random Clustering (RandC) corresponds to a clustering of the *link table* in which the link tuples are in random order with the exception that links of the same *origin* node are clustered together. Random Clustering is included in our experimental evaluation as the straw-man to determine the path query processing cost when no clustering strategy is deployed. In this paper, we compare its

performance in path query processing with those of all other clustering strategies.

# 5 Extended Clustering Strategies

## 5.1 Combining Spatial Partitioning with Swapping (Hybrid)

One of the techniques commonly used in min-cut graph partitioning algorithms is node swapping (see Section 3.1.1 for a detailed discussion). We develop a hybrid clustering approach that combines the spatial partition clustering (Section 2.2) with the swapping technique (Section 3.1.1). We call it the hybrid approach. This hybrid approach starts by performing the spatial partition clustering on the graph, striving as before to reach high occupancy rate for each partition. In contrast to the SPC approach where the page occupancy rate is approximately 100%, we allow the minimum occupancy rate to be as low as 90% for the hybrid approach[4]. This relaxation on the occupancy rate leaves some space on each page that can be used for more effective single-node swapping.

When the spatial partitioning process is complete, we perform single-node swapping with the maximum occupancy rate set to 100%. This means that swapping cannot fill a partition so as to exceed the size of a page. We conduct such swapping for several iterations with the partitioning objective based on minimum number of inter-partition links. Next, we continue by performing pair-wise swapping for several more iterations. This is designed to give the partitions that are full a chance to swap nodes with other partitions to reach a better cut. In contrast to the swapping in the two-way partitioning approach in which each node has only one other partition to swap to (see Section 4.1), the swapping in our hybrid approach has to consider all other partitions a node can potentially swap to. This is because the spatial partitioning process creates many initial partitions. It is to be expected (and our experiments confirm) that the resulting clustered *link table* has a few more pages than the SPC-clustered *link table* because of its slightly lowered occupancy rate, but the number of inter-partition links could potentially be reduced by the swapping process.

## 5.2 Two-Way Partitioning Based on Link Weight (TWPC_wgt)

In applying the two-way partitioning algorithm to graph clustering, our objective is to reduce the number of cross-page links (TWPC_con in Section 3.1). We believe such an approach will lead

[4]Our experiments showed that 90% is a good compromise between high occupancy rate and sufficient room for subsequent node swapping.

graph-traversal path computation to traverse more intra-page links and less inter-page links since the numbers of the latter are reduced. As a result, the page misses happening during path search are likely to be reduced. We now extend the partitioning objective to include link weights also.

As before, we set our objective to first also minimize the number of total cross-page links. However, if two or more possible cuts have the same number of reductions in terms of inter-partition links, we break the tie by giving the swapping priority to the cut that results in the maximum sum of weights of all cross-page links[5]. This objective is based on the fact that our path search algorithm, the *Dijkstra* algorithm, is a priority search that gives priority to the node with the minimum traversed weight so far. With everything else being equal, a link with a larger weight is less likely to be favored by the *Dijkstra* algorithm to be traversed next than a link with a smaller weight. Therefore, given an equal number of cross-page links, it can be expected that a partition that has a larger total weight of all cross-page links could potentially further improve the I/O efficiency of path query computation based on the *Dijkstra* algorithm. Note that to reduce the number of cross-page links is still the first priority because if we put the maximum sum of all weights of all cross-page links as the first priority, the resulting cut will possibly have a large number of cross-page links, making the partitioning ineffective.

# 6 Testbed Environment

In this section, we first discuss our experimental testbed setup, followed by the graph representation, and data sets, and then experimental parameters and measurements.

## 6.1 Experimental Testbed Setup

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++.

[5]Note that for stable graph clustering that we consider in this paper, the link weight used in this partitioning objective should not be an unstable link attribute that changes frequently.

### 6.2 Graph Representation

We use the *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). Because 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 must store relevant link attributes in their corresponding link tuples. In this paper, the link tuple adopted in our experiments is set to 128 bytes. The reader can find a listing of possible attributes that could be associated with such links. As discussed in Section 1, depending on the type of query, the link attributes must be kept with the link itself in order to allow for the filtering of links from the candidate paths during path processing, such as for the spatial constraints in query **Q3** from Section 1.

### 6.3 Experimental Transportation Networks

We test two kinds of graphs: randomly generated graphs and a real (fine-granularity) network representing the streets of Ann Arbor City that has 5,596 nodes and 14,033 links. We experiment with random graphs with 5,000 nodes and vary the average out-degree from 2 to 8. To create a random graph with average out-degree $d$, we randomly select, for each node, from 1 to $2 \times d - 1$ outgoing links and, for each link, we randomly select a *destination*. The *destination* must be different from the *origin* of the same link, and the *destinations* of two different outgoing links from the same *origin* must be different. The *weight* for each link is chosen to be a random integer between 1 and 100. We also create two sets of such random graphs, one with high locality, and the other with no locality. To control the locality of a random graph, we associate each node with an x-coordinate and a y-coordinate value. For graphs of high locality, we allow a link to exist only when its *origin* and *destination* are within a relatively close vicinity as compared to the total area. For graphs of no locality, we set no such limitation.

The reason we experiment with random networks of both high and no locality is because more advanced GIS applications such as Intelligent Transportation Systems need to model graphs beyond the road transportation networks (such as the Ann Arbor city network). For example, the graphs of airline flight routes exhibit no planarity and locality therefore can be better modeled by random graphs with no locality. An inter-modal network of both subway train and bus routes, however,

exhibits high locality without planarity, therefore can be modeled by random graphs with high locality.

### 6.4 Clustering and Path Search Algorithms

In our experiments, we first prepare the network data using the various clustering techniques proposed in this paper, namely SPC (Spatial Partition Clustering), Hybrid (the hybrid approach that combines SPC and node swapping), TWPC_con (Two-Way Partition Clustering based on connectivity), TWPC_wgt (Two-Way Partition Clustering based on stable link weights), TopoC (Topological Clustering), and RandC (Random Clustering). The data then is layed out on disk based on this preprocessing stage.

Thereafter, for each experiment, we apply the Dijkstra algorithm to conduct a single-source shortest path search for randomly selected source nodes $i$ to all other nodes in the network. Such computation corresponds to the graph-traversal search for the shortest-path from node $i$ to the one node $j$ that is the farthest away from $i$. Hence this set of experiments tests the worst-case scenario in searching a shortest path from $i$.

### 6.5 Settings of Parameters

In this paper, the size of pages on disk and in the main memory buffe is set to be 4 K bytes each. The experiments are based on a buffer containing up to 240 pages, i.e., varying the size of the buffer from 64 K bytes up to 960 K bytes. The size of the entire *link table* is about 2 M bytes, with a small difference between the various clustering techniques.

Although the experiments presented in this paper are based on buffer sizes up to 960 K bytes, the adequate buffer size for path query processing is proportional to the size of the underlying network. The experimental networks in this paper (i.e., the Ann Arbor city network) are of medium sizes (Section 6.3). The sizes of larger cities can be many times larger. For example, the Detroit road network we are using for related research in this project has more than 50,000 links which is about three times the size of the Ann Arbor map. Consequently, the buffer requirement should increase for larger maps. Second, resolving constraints embedded in the path queries may incur heavy I/O activities which takes away the buffer space from the path search process. Lastly, in a multi-user and multi-tasking database system, one cannot assume that the entire resources such as the buffer

space are available to one single query process. We therefore are motivated to find the clustering strategy that can process path queries efficiently while using as small a portion of the buffer as possible. Therefore, in reality, the buffer requirement for processing constrained path finding on a large network in a multi-user database environment can be many times larger than the various buffer sizes depicted in our experimental evaluation in Section 7.

Given that we use the same search algorithm Dijkstra in our experiments, the CPU processing costs are all fairly comparable whereas the number of disk pages that must be transferred between the slower secondary storage device to the faster main memory system for processing, referred to number of page input/output operations or in short I/Os, varies significantly based on the placement of link tuples on pages. In most systems, I/Os are on the order of 100-fold more expensive than CPU operations, and hence the critical factor for determining system performance are the I/O and not the CPU costs. Hence, we measure performance in our work in terms of number of I/Os, with the actual total processing time being a multiplicative of this I/O count based on the average I/O cost in the given system.

# 7 Experimental Evaluation

## 7.1 Experiments on Ann Arbor Road Network

In the first set of experiments, we use the Ann Arbor road network and conduct single-source shortest path search for randomly selected nodes using the clustering techniques proposed in this paper.

In the first set of experiments, we use the Ann Arbor road network. The results in Figure 7 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 the Random clustering is very high, making it hard to see the difference in performance between the other five clustering approaches, we plotted Figure 8 without showing the Random clustering results. In Figure 8, it is clear that SPC performs the best, followed by, in exact order, TWPC_wgt, TWPC_con, Hybrid, and TopoC. It is surprising to see that although TopoC has the worst performance among the five clustering optimizations, it is still much more effective than RandC. This is contradictory to the suggestion in [37, 40] that topological clustering is not effective for highly cyclic graphs such as our road network.

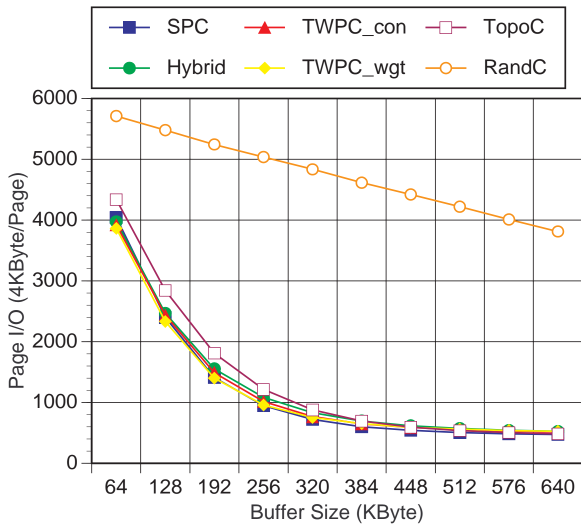


Figure 7: I/O Cost of Searching the Longest Path on Ann Arbor Map.

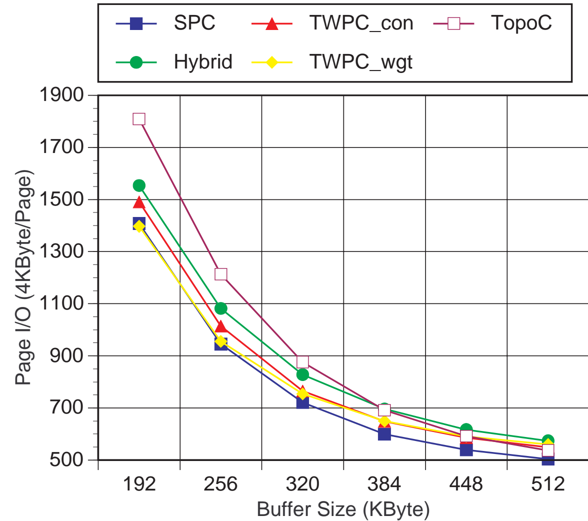


Figure 8: I/O Cost of Searching the Longest Path (Excluding Random Clustering) on Ann Arbor Map.

It is also interesting to note that the Hybrid approach performs worst than the SPC. This indicates that the partitioning objective we set to minimize the cross-page links during the swapping process may not be as relevant to highly interconnected near-planar graphs like the Ann Arbor network as spatial proximity which SPC is based upon. This also helps to explain why SPC performs better than TWPC_wgt and TWPC_con. The fact that TWPC_wgt performs better than TWPC_con indicates that by incorporating link weights, the partitioning objective of TWPC_wgt catches the expansion behavior of the *Dijkstra* algorithm better than that of TWPC_con. Note that when the buffer size is greater than 512 K bytes, all five clustering strategies perform the same. This is because the size of the buffer is large enough to contain the expansion locality of the *Dijkstra* algorithm captured by all five clustering optimizations. Therefore, roughly one pass for such a large buffer would be sufficient to compute the single-source shortest paths.

### 7.2 Experiments on the High-Locality Random Graphs

The second set of experiments is based on a randomly generated graph with 5,000 nodes, average out-degree of 3, and high locality. We conduct the same single-source path search experiments described above. While the Ann Arbor network is very planar and interconnected, the high-locality random graph does not guarantee planarity and high interconnection. The results in Figure 9 show that RandC remains the distant worst, with the TopoC significantly worse than the other four

clustering approaches.

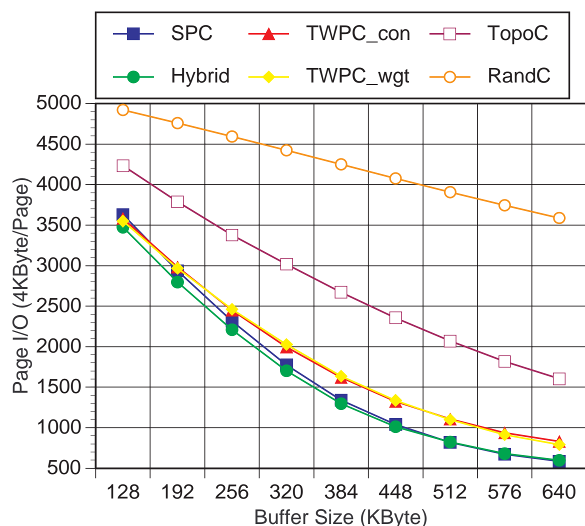


Figure 9: I/O Cost of Searching the Longest Path on the High-locality Random Graph.

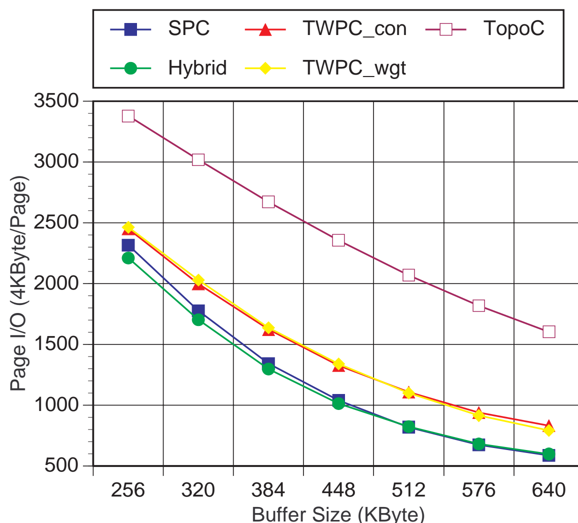


Figure 10: I/O Cost of Searching the Longest Path (Excluding Random Clustering) on High-locality Random Graph.

The close-up results in Figure 10 show that the Hybrid and SPC perform better than TWPC_con and TWPC_wgt, with the Hybrid having a slight edge. This is different from the experimental results on the Ann Arbor network where the Hybrid is worse than the other three. This indicates that the partitioning objective of minimizing cross-page links does help in bringing down the I/O cost incurred by the *Dijkstra* path search for high-locality graphs without high interconnection and planarity. The superior performance of both the Hybrid and SPC over the TWPC_con and TWPC_wgt indicates that each partition created by our proposed SPC partitioning algorithm is tailored to fit perfectly into a buffer page. Therefore the resulting graph partitions are better, in terms of both page occupancy rate and expansion locality exhibited by the *Dijkstra* algorithm, than the partitions created by the TWPC approaches that use *divide-and-conquer* to distribute partitions into pages.

## 7.3 Experiments on the Low-Locality Random Graphs

In the third set of experiments, we test a randomly generated graph with 5,000 nodes, average out-degree of 3, and no locality. Interestingly, the results in Figure 11 show that RandC and SPC are equally the worst. This can be explained by the fact that without locality, the spatial proximity is irrelevant for proper partitioning. Consequently, the SPC performs the same as the

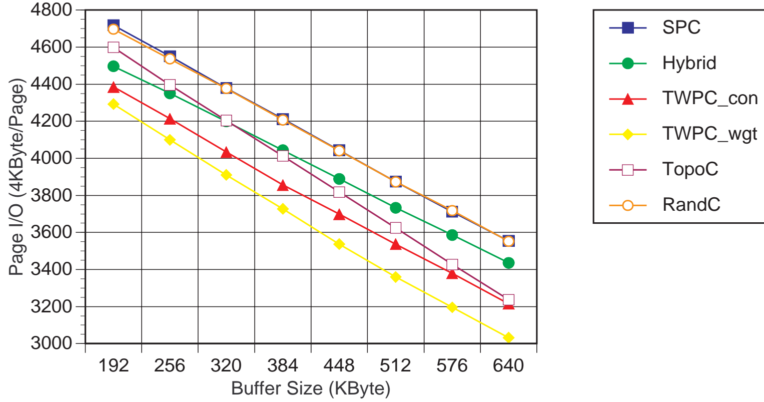


Figure 11: I/O Cost of Searching the Longest Path on the Low-locality Random Graph.

RandC on graphs with no locality. The swapping process in the Hybrid approach tries to correct the irrelevant spatial partitioning, but its performance is still worse than the TWPC approaches, and even worse than the TopoC for some buffer sizes. The results also show that TWPC_wgt has the best performance, followed by TWPC_con. This indicates that the two TWPC approaches are not locality-dependent, therefore have better performance than the SPC and the Hybrid approaches on graphs with no locality. The link-weights based partitioning objective in TWPC_wgt that we have proposed is an effective optimization over the pure connectivity based objective in TWPC_con. We note however that the TWPC_wgt has the limitation that the link weight used as the partitioning objective must be stable.

### 7.4 Experiments on Paths of Different Length

So far, our experiments focus on the worst-case scenario, i.e., the cost of computing the longest among all shortest-paths to all possible destinations. We now explore the path search performance on paths of different length, namely short paths, medium paths, and long paths. We define the direct distance between the farthest node-pair as $d_{max}$, and the direct distance between the two end nodes of a path as $d_{short}$ for short paths, $d_{medium}$ for medium paths, and $d_{long}$ for long paths. Then the following relations hold:

$$d_{short} < (d_{max}/3), \text{ and}$$
$$(d_{max}/3) \leq d_{medium} < (d_{max} \times 2/3), \text{ and}$$
$$d_{long} \geq (d_{max} \times 2/3)$$

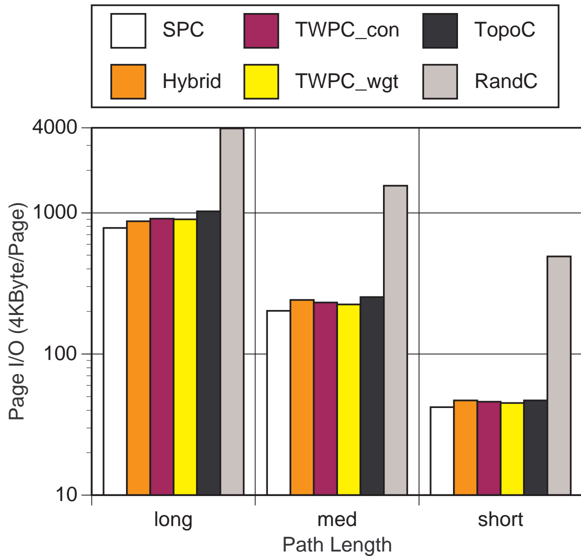


Figure 12: I/O Cost By Length of Paths on the Ann Arbor Network.

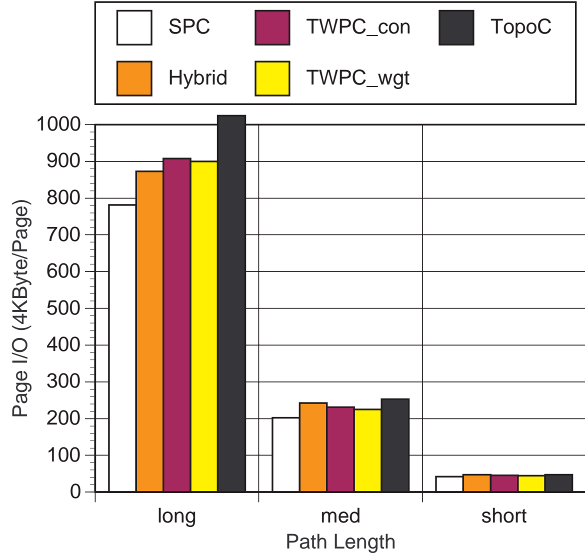


Figure 13: I/O Cost By Length of Paths (Excluding Random Clustering) on the Ann Arbor Network.

Because such a direct-distance based classification is only meaningful if the graph has locality and is highly inter-connected, we conduct this set of experiments on the real Ann Arbor city map only.

We randomly select a number of node-pairs from each category, conduct path search, and collect the average results. The buffer size is set to 256 K bytes. Because the performance for RandC is much worse than any other clustering approach, we use the log scale in Figure 12. Figure 13 shows the results in linear scale without RandC. In Figure 13, we can see more clearly that SPC consistently has the best performance for all three kinds of paths, while TopoC generally has the worst performance.

## 7.5 Experiments on Average Out-degree

In this last set of experiments, we randomly generated graphs with 5,000 nodes, varying the average out-degrees from 2 to 8. We set the buffer size to 960 K bytes in order to accommodate graphs with large average out-degree. The purpose of the experiments is to find out which clustering strategies work better for graphs of various average out-degrees. Because graphs of high out-degrees automatically lose locality with evenly distributed nodes, we only generate graphs with no locality for this set of experiments.

The experimental results in Figure 14 show that both TWPC_wgt and TWPC_con perform

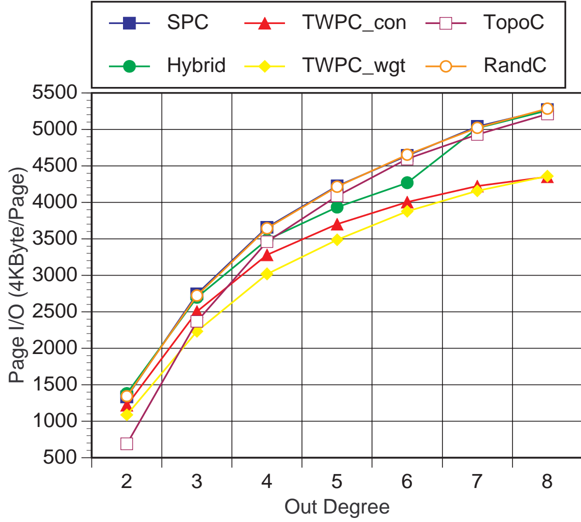


Figure 14: I/O Cost By Average Out-degree on the Random Graphs.

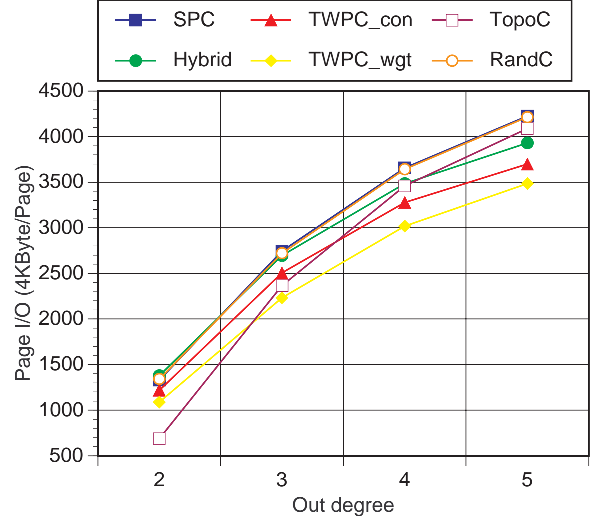


Figure 15: I/O Cost By Average Out-degree on the Random Graphs (Close-up).

better as the average out-degree increases. Although TWPC_wgt has a better performance than TWPC_con, the performance curve of TWPC_con actually becomes (i.e., drops) more favorably as the average out-degree increases. The other four clustering approaches all fair poorly as the average out-degree goes up. Figure 15 is a close-up of Figure 14 with average out-degrees from 2 to 5. In Figure 15, the performance of TopoC is much better than the others when the average out-degree is 2. This is because when the average out-degree is small, there are primarily acyclic components in the graph. The TopoC strategy is extremely efficient for the acyclic graphs and therefore performs much better when there are many acyclic subgraphs in the graph.

# 8 Conclusions

Efficient path query processing over transportation networks is important for many GIS applications. In this paper, we consider the optimization of path query processing using graph clustering techniques. Clustering optimization is attractive because it does not incur any run-time cost, nor does it require auxiliary data structures that demand memory. More importantly, it is complimentary to many of the existing path query solutions typically at the data structure or at the algorithmic level.

In this paper, we first propose a new clustering technique, called the spatial partition clustering (SPC), for path query optimization for transportation networks. Next, three other graph clus-

tering techniques, namely the two-way partitioning clustering, topological clustering, and random clustering, are identified from the literature, fine-tuned for GIS path query optimization, and then implemented in our uniform testbed. In addition, based on the spatial partition clustering and the two-way partitioning clustering, we develop two extensions, called the hybrid approach and the two-way partitioning based on link weight respectively.

This paper presents an extensive experimental evaluation of the comparative performance of the above six graph clustering techniques. Our experiments are based on three kinds of networks. They are the Ann Arbor city road map; randomly generated graphs with high locality modeling GIS maps such as inter-modal bus and subway routes; randomly generated graphs with no locality modeling GIS networks such as airline routes. The experiments are conducted by varying testing parameters such as memory buffer size, path length, locality, and average out-degree. The experimental results show that our spatial partition clustering performs the best for the real network; the hybrid approach has the best performance for random graphs with high locality; whereas the two-way partitioning based on link weights works the best for random graphs with no locality. Such experimental results are important, representation a foundation for establishing guidelines in the selection of the best clustering technique based on the type of network prevalent in a given application.

For future work, effective graph clustering techniques can also be extended to solve more general path problems such as recursive query processing. Results from this paper can also be exploited for further optimization of complex path query processing with embedded constraints. For instance, our exploration of a framework of spatial path queries [18] is based on the SPC solution first introduced in this paper. Lastly, clustering techniques could be explored to take into account knowledge about in which s spatial location paths with certain properties can be found. Such knowledge could be utilized to constrain the search and also to adjust the clustering of links for origin-destination pairs that meet this particular class of queries.

## 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 Jagadish, H. V., "Efficient Search in Very Large Databases," *Proc. of the 14th VLDB Conf.*, Los Angeles, California, 1988, pp. 407 – 418.

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

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

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

[6] 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.

[7] 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

[8] 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.

[9] Brinkhoff, T., Kriegel, H., Schneider, R., and Seeger, B., "Multi-Step Processing of Spatial Joins", *Proc. of the 1994 ACM SIGMOD Int. Conf. on Management of Data,* 1994, pp. 197 – 208.

[10] Carey, M.R., Johnson, D.S., and Stockmeyer, L., "Some Simplified NP-Complete Graph Problems," *Theoretical Computer Science,* 1976, pp. 237 – 267.

[11] 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.

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

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

[14] 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.

[15] M. F. Goodchild, "Tiling large geograhical databases", In A. Buchmann, O. G—nther, T.R. Smith and Y.-F. Wang, editors, Design and Implementation of Large Spatial Databases. New York: Springer-Verlag, 137-146. 1990.

[16] M. F. Goodchild, and Y. Shiren, A Hierarchical Spatial Data Structure for Global Geographic Information Systems, Computer Vision, Graphics and Image Processing: Graphical Models and Image Processing 54 (1): 31-44. 1992.

[17] Huang, Y. W., Jones, M.C., and Rundensteiner, E.A., "Symbolic Intersect Detection: A Method for Improving Spatial Intersect Joins," *Journal of GeoInformatica*, Special issue on Spatial Database Systems, 2:2, 149-174 (1998), Kluwer Aca. Pub.

[18] Huang, Y. W., Jing, N. and Rundensteiner, E., "Integrated Query Processing Strategies for Spatial Path Queries." *IEEE Int Conf. on Data Engineering*, 1997: pp. 477-486.

[19] Huang, Y. W., Jing, N. and Rundensteiner, E., Yun-Wu Huang, Matthew C. Jones, Elke A. Rundensteiner: "Improving Spatial Intersect Joins Using Symbolic Intersect Detection." *SSD Conf.*, 1997: pp. 165-177.

[20] Huang, Y. W., Jing, N. and Rundensteiner, E., "A Cost Model for Estimating the Performance of Spatial Joins Using R-trees." *SSDBM*, 1997: pp. 30-38.

[21] Huang, Y.W., Jing, N. and Rundensteiner, E.A., "A Hierarchical Path View Model for Path Finding in Intelligent Transportation Systems," *Journal of GeoInformatica*, 1:2, pp. 125-159 (1997), Kluwer Aca. Pub..

[22] Huang, Y. W., Jing, N. and Rundensteiner, E., "Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations" *VLDB* 1997, pp. 396-405.

[23] Huang, Y.W., Jing N. and Rundensteiner, E. A., "Query Processing Strategies for Spatial Path Queries," *IEEE Int. Conf. on Data Engineering, (ICDE-13)*, April 1997, England.

[24] Huang, Y.W., Jing, N., and Rundensteiner, E. A., "Path View Algorithm for Transportation Networks: The Dynamic Reordering Approach," *ACM Workshop on Geographic Information Systems* (ACM GIS'96), Washington, D.C., Nov. 1996.

[25] 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.

[26] Huang, Y. W., Jing, N. and Rundensteiner, E., "Effective Graph Clustering for Path Queries in Digital Map Databases," *Proc. 5th Int'l Conf. CIKM*, 1996, pp. 215 – 222. Washington D.C., Nov., 1996.

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

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

[29] 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.

[30] Jing, N., Huang, Y.W., and. Rundensteiner, E.A., "Hierarchical Encoded Path Views for Path Query Processing: An Optimal Model and Its Performance Evaluation", *IEEE Transactions of Knowledge and Data Eng.,*, 10(3): 409-432 (1998).

[31] Jing, N., Huang, Y. W., and Rundensteiner, E., "Hierarchical Optimization of Optimal Path Finding for Transportation Applications," *Proc. 5rd Int'l Conf. CIKM*, 1996. pp. 261-268.

[32] 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.

[33] 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.

[34] Preparata, F.P. and Shamos, M.I., "Computational Geometry," Springer, 1985.

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

[36] Shamos, M.I. and Hoey, D.J., "Geometric Intersection Problems," *Proc. 17th Annual Conf. on Foundations of Computer Science*, 1976, pp. 208 – 215.

[37] Shekhar, 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.

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

[39] Zhan and Noon, Shortest path algorithms: an evaluation using real road networks, *Transportation Science* 32, pp. 65-73, 1998.

[40] Zhao, J.L. and Zaki, A., "Spatial Data Traversal in Road Map Databases: A Graph Indexing Approach," *proc. 3rd Int'l Conf. CIKM*, 1994, pp. 355 – 362.