Efficient Search of Path-Constrained Moving Objects

by

Bradford G. Nickerson and Thuy Thi Thu Le
Faculty of Computer Science
University of New Brunswick. Canada

Technical report TR08-191, September 08, 2008

Faculty of Computer Science

University of New Brunswick

Fredericton, N.B. E3B 5A3

Canada

Phone: (506) 453-4566

Fax: (506) 453-3566

Email: fcs@unb.ca

www: http://www.cs.unb.ca

Abstract

We present a spatio-temporal data structure called the Graph Strip Tree (GStree) for indexing objects constrained to move on a planar graph. The GStree is designed to efficiently answer range queries about the current or past positions of moving objects. To test the efficiency of our data structure, a road network of 66,437 roads was used. A set of 443,983 moving objects was randomly positioned on edges of the planar graph, with location updates made over 5, 50 and 100 time steps. This resulted in $3.2 \cdot 10^6$, $32 \cdot 10^6$ and $64 \cdot 10^6$ location updates, respectively, indexed by the data structures. Average search times for random queries to find moving objects indexed by a GStree were compared to average search times for the same queries on moving objects indexed by a MON-tree. Results indicate that the GStree is up to 24 times faster than the MON-tree for internal memory searching, and visits between 3.6 and 38 times fewer nodes. Analysis indicates the GStree will be significantly faster for external memory search where the search time is dominated by the number of disk I/Os.

1 Notation

A = number of kinetic events in the time interval $[t_{now}, t_q]$ or $[t_q, t_{now}]$
B = disk block size
C_i = leaf node of a graph strip tree containing a root bounding box for strip tree i and a pointer to the corresponding interval tree I_i
D = average number of visited nodes for a query
D_i = list of moving objects from edge e_i intersecting a query rectangle R
e_i = one edge in the planar graph representing the underlying network
E = number of edges in the underlying network planar graph
$\epsilon_x^1 = \max_t(|\hat{f}_x(t) - f_x(t)|)$ = maximum deviation for the x dimension
$\epsilon_y^1 = \max_t(|\hat{f}_y(t) - f_y(t)|)$ = maximum deviation for the y dimension
$f_x(t)$, $f_y(t)$ = object path in the xy-plane
$\hat{f}_x(t)$, $\hat{f}_y(t)$ = polynomial approximation of an object path in the x, y plane
F_i = list of intervals of one edge e_i intersecting a query rectangle R
G = planar graph containing moving objects
H = total time interval that queries can be issued for
I = interior node of a tree
I_i = interval tree representation of a graph edge e_i
K = number of points reported by a query
$k = \lceil K/B \rceil$ = number of disk blocks to store the output reported by the query
L = set of sorted intervals (either left L_L or right L_R) in an interval tree
m = number of time intervals in $[0, T]$
M = maximum number of children of a multi-way search tree
M_i = interior node of a graph strip tree (GStree)
N = number of moving objects
$n = \lceil N/B \rceil$ = number of disk blocks to store all N moving objects
$p_i(t)$ = position of p_i at time t
$Q_1 = (R, t_q)$ = query type 1; find moving objects intersecting rectangle R at time t_q

$Q_2 = (R, [t_1, t_2])$ = query type 2; find moving objects intersecting rectangle R at any time during time interval $[t_1, t_2]$
$r \in [0, 1]$ = position of a point along a polyline
$S = p_1, p_2, ..., p_N$ = set of moving points in the xy-plane
S_i = strip tree representation of a graph edge e_i
t = time
t_b = begin time of a trajectory falling into one spatial cell
t_e = end time of a trajectory falling into one spatial cell
t_{now} = current time
t_{ref} = index creation time
t_q = query time
T_i^j = external interval tree for moving objects in time interval $[t_{j-1}, t_j]$ on graph edge e_i
UI = update time interval
v_i = one vertex in the planar graph representing the underlying network
V = number of vertices in the network planar graph
W = limit on how far queries can ask into the future
$[0, T]$ = time domain

2 Introduction

A significant challenge in spatial and spatio-temporal databases is how to improve the response time for query processing of moving objects, called continuous query processing (e.g. [17], [15]). Saltenis et al [18] divide the problem of indexing the positions of continuously moving objects into two categories. Queries about the current and anticipated future positions of moving objects define one category. Such queries are likely to be used in real-time and near real-time systems. Applications such as traffic control, emergency response and navigation while driving fall into this category. The second category focusses on the history of the positions of moving objects. Queries on historical data are likely to be used in applications such as planning, event reconstruction and training. Our research addresses the latter category.

Mokbel et al [14] describe three main types of continuous queries. They are (1) moving queries on stationary objects, (2) stationary queries on moving objects, and (3) moving queries on moving objects. In addition, queries can take the form of range queries (e.g. circular or rectangular) or k-nearest neigbhor queries. For example, when the query is issued from a vehicle moving on a highway, the following queries are examples of nearest neighbor queries of type (1): *Which is the nearest gas station now?* or *Based on my current direction and speed of travel, show me the nearest two gas stations in the next five minutes?* Answers for such questions require the location of the moving object at the query time and a good location prediction technique for the moving object. The answer changes continuously as the object issuing the query moves.

An example of a continuous range query of type (2) might be *How many moving vehicles will be in the center of the city ten minutes from now?* Depending on the size of the query, such a counting query might return an answer in the hundreds or thousands. The accuracy of the returned answer depends on the accuracy of prediction of future positions of moving objects, which depends, in turn, on the accuracy of the model of the underlying road network on which the vehicles are travelling.

Two basic approaches are used to index the moving objects. Indexing the trajectories of the objects, with updates of trajectories triggering index updates, permits storage and indexing of the paths of objects described as a function $p_i(t)$ of time t for moving point i. This is the approach followed by Vazirgiannis and Wolfson [20] and by Agarwal et al [2], so that a feasibly sized index can be built for the (potentially) many moving objects. The second approach considers updates arriving at regular intervals for all objects. Hadjieleftheriou et al [13] follow this approach. Regular interval updates simplify the update algorithm, but require more space to store object positions that may be a linear extrapolation of the two previous object positions.

3 Problem Definition

Our paper addresses indexing the history of the positions of moving objects whose positions are updated on a regular basis. This corresponds to the second catgory mentioned in the Introduction.

In addition, our data structure supports stationary queries on moving objects. We assume there are $m + 1$ positions for each moving object defined on m equally spaced time intervals in the time domain $[0, T]$. In addition, we assume that the object positions are restricted to move on a planar graph (possibly disconnected) defined by its edges E and vertices V. Figure 1 shows a small part of a test road network. We support two types of queries; time instant queries defined as $Q_1 = (R, t_q)$ to find the K moving objects intersecting rectangle R at time t_q, and time interval queries defined as $Q_2 = (R, [t_1, t_2])$ to find the K moving objects intersecting rectangle R at any time during time interval $[t_1, t_2]$. Both query types can be counting queries (report only K) or reporting queries (report the identity of the K moving objects satisfying the query).

4 Related Work

Since the work on persistent data structures by Driscoll et al [11], there has been significant research into indexes able to store complete histories of data repositories. Kinetic data structures [6] make search of complete histories of moving objects possible by updating the data structure only when significant kinetic events occur. For example, when a vehicle moving on a road network reports a position update, this can be considered a significant event causing the insertion of a new trajectory for the updated vehicle. An excellent summary of known index methods for moving points up to the year 2002 or so is contained in Agarwal et al [2] and the references therein. A recent paper by Ni and Ravishankar [15] contains a good overview of data structures experimentally validated for moving object indexing. We have summarized the characteristics of some of the known approaches for indexing moving objects in 1.

Table 1 differentiates data structures based on their support for future time queries and whether or not they assume that the objects being indexed are constrained to move on an underlying graph. If movement is restricted to a planar graph, then the index should be able to take advantage of this to achieve less storage than would be required if objects were free to move anywhere in space. As we will show, the GStree is designed to exploit this constraint.

The time-parameterized R-tree (TPR-tree) of Saltenis et al [18] was one of the first data structures to answer queries about future moving object positions. Moving objects are represented as

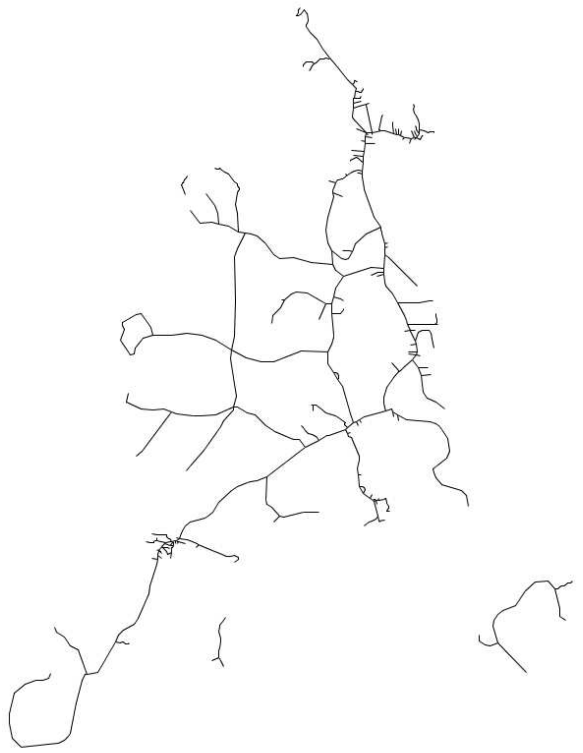

Figure 1: Part of a road test network consisting of 257 polylines in Grand Manan Island in the very southern part of the Province of New Brunswick, Canada.

Table 1: Different data structures for indexing moving objects. Here H = support for queries on the history of moving objects, F = support for future time queries, C = movement constrained to a planar graph, L = movement constrained to be piecewise linear, E = experimental validation.

Name	H?	F?	C?	L?	E?	
TPR-tree	N	Y	N	Y	Y	[18]
partition tree	N	N	N	Y	N	[2]
kinetic range tree	N	Y	N	N	N	[2]
SETI	Y	N	N	Y	Y	[7]
MON-tree	Y	N	Y	Y	Y	[9]
MVR-tree	Y	N	Y	Y	Y	[13]
PA-tree	Y	N	N	N	Y	[15]
GStree	Y	N	Y	Y	Y	this paper

reference points with a velocity vector at index creation time t_{ref}. Time-parameterized rectangles bounding the positions of all moving points or other bounding rectangles are stored in R*-tree interior nodes. Answering a query $Q_1 = (R, t_q)$ to find moving objects inside rectangle R at time t_q requires computing specific bounding rectangles for R-tree nodes at time t_q. Moving objects are updated when a new position is obtained (which may also indicate a new velocity). Each update deletes the updated reference point and inserts (using minimal overlap of the integral of area of intersection of time-parameterized rectangles) the new point, updating interior R*-tree bounding rectangles appropriately.

The simulations reported by Saltenis et al [18] run for 600 time units (minutes) with an average time interval of $UI = 60$ time units between updates of moving object positions. An average of four search queries per time unit are issued, giving a mixed workload of 2,400 queries plus the 1,000,000 updates of 100,000 simulated moving objects. Their experimental results indicate that the TPR-tree is approximately an order of magnitude faster (in terms of number of disk I/O operations) than the R*-tree for searching for the experimental data and workload used. For values of $W \in [0, 40]$ time steps, where W is the limit on how far queries can ask into the future, minimum search times are obtained when H is between $UI/2 + W$ and $UI + W$. H is the total time interval that queries can be issued for.

Agarwal et al [2] present a data structure for indexing moving points that uses $O(N/B)$ space, for $N =$ the number of moving points and $B =$ the disk block size. Their result based on partition trees answers a Q_1 or Q_2 query using $O((N/B)^{1/2+\epsilon} + k)$ I/Os in the worst case, where $k = \lceil K/B \rceil$, K is the number of moving objects reported and ϵ is a small positive constant. They coin the term *time-oblivious index* for their approach as the index only needs to be updated when the trajectory of a point changes. This partition-tree based index assumes that the points move in a straight line with constant velocity. Updates on their moving point index requires $O(\log_B^2 n)$ expected I/Os. Their data structure uses a primary partition tree for points representing the dual space of the (x, t) projections of the lines representing paths of moving objects. Secondary partition trees indexing point subsets from the dual of the (y, t) projections of the moving object paths are placed strategically within the primary partition tree. Suitable choices for fanout of the primary partition tree and which nodes to attach secondary partition trees to leads to the claimed space and time bounds.

Using what they call a multiversion external kinetic range tree, Agarwal et al [2] show how to answer Q_1 queries in $O(\log_B n + k)$ I/Os using $O(n \log_B n/(log_B log_B n))$ disk blocks. Their data structure requires a multiversion catalog structure at nodes of a priority search tree, and two multiverison kinetic B-trees for tracking when x or y coordinates coincide, leading to two deletions and two insertions corresponding to the swap in order of two moving points. An event queue stored in a B-tree keeps track of when the next kinetic event (change in order of two moving points or change in trajectory of one moving point) occurs. Agarwal et al also introduce the notion of a query that is a monotonically increasing function of $|t_q - t_{now}|$, where t_{now} is the current time. This data structure requires $O((A/n)^{1/2} n^{\epsilon} + k)$ expected I/Os, where $0 \leq A \leq \binom{n}{2}$ is the number of kinetic events in the time interval $[t_{now}, t_q]$ or $[t_q, t_{now}]$. This data structure attaches a multiversion kinetic B-tree at every node of a grid tree. To achieve the expected I/Os bound, the moving point positions are assumed to be drawn from a uniform random distribution.

SETI, a Scalable and Efficient Trajectory Index [7], has a three-part indexing structure. An in-memory part called the *front-line* uses a hash table to store the last known position of all objects being indexed. An object position update triggers generation of a new line segment starting at the last known position and ending at the updated position. This line segment is inserted into a spatial cell index structure (that splits the line segment at cell boundaries) and the corresponding temporal index. In their paper, Chakka et al [7] use a uniform rectangular grid for defining spatial cells. Each spatial cell has its own temporal index. The new position replaces the previous one in the front-line. The temporal index contains *lifetime* values (t_b, t_e) for all line segments on a *data page* falling into this spatial cell in the time interval $[t_b, t_e]$. The data page lifetimes are indexed using an R*-tree. When the index is searched with a $Q_2(R, [t_1, t_2])$ query, the temporal indices of all spatial cells intersecting R are searched to find data pages containing segments of trajectories $\in [t_1, t_2]$.

As Chakka et al [7] point out, the fact that moving objects fill data pages in chronological sequence means that the R*-tree leaves of the temporal index have very little overlap, giving good discrimination for the temporal part of the search. In their experimental evaluation with 1,000 moving objects consisting of 4,000,000 segments (with uniform random initial locations and movements), SETI required about 1/40 of the space to store the index compared to a 3-dimensional R-tree. Q_2 queries on the same data with 0.1% of the data in range showed SETI requiring less than half the I/Os (on average) compared to the TB-tree (Trajectory Bundle tree) [16]. In addition, this testing showed SETI requiring less than 1/3 the overall time for this type of query (on average) compared to the TB-tree. The SETI spatial cell subdivision was experimentally found to be optimal (for the test data used) when the number of cells was 400 (20 by 20 rectangular cells).

In tests with 1,000 objects (modelling cell phone users) moving on the road network of San Joaquin county, a 0.1% of data in range Q_2 type query took (on average) about 1/2 the time using SETI compared to using the TB-tree. A second test simulated up to 9,200 objects moving on the San Joaquin county road network for 8 hours, then stopped for 16 hours for each of 30 days (what they call *long update interval* testing). On average, SETI answered 0.1% type Q_2 queries nine times faster than the same data indexed by a TB-tree. Chakka et al [7] compare SETI to the TB-tree with up to 160,000 moving objects, each having 100 segments. With this larger number of moving objects, SETI was able to answer 0.1% type Q_2 queries about eight times faster (on average) than the TB-tree. The significant advantage of SETI is due to the separate R*-tree temporal indices for each spatial cell that give very little overlap at the leaf level.

The Moving Objects in Networks (MON) tree [9] can represent moving objects on a planar graph of edges consisting of polylines, or on routes consisting of sets of edges from a planar graph. If one assumes that moving objects have a constant velocity along a route, the route oriented model leads to fewer entries in the index data structure. The MON-tree data structure consists of two R-trees and a hash table. The top 2-dimensional R-tree has leaf nodes corresponding to one polyline (or one set of polylines for the route model) from the planar graph. Each leaf node of the top R-tree points to a bottom 2-dimensional R-tree that indexes all moving objects on this polyline or route. The two dimensions indexed by the bottom R-tree are position $r \in [0, 1]$ and time $t \in [0, T]$. A leaf node of the bottom R-tree stores a rectangle of one moving object's position interval $[r_1, r_2]$ corresponding to the time interval $[t_1, t_2]$ when it was moving along this polyline (or route). A separate hash table with polyline (route) number as the key is used on inserting a new position, time interval for a moving object. The hash table points to the bottom R-tree for an existing polyline (route). This avoids visiting the top R-tree, and proceeds immediately to the appropriate moving object index.

De Almeida and Güting [9] compare the MON-tree to the Fixed Network R-tree (FNR-tree) [12]. The planar graph chosen for testing was a German road network consisting of 4,273 edges (995 routes), and data sets with up to 50,000 moving objects (on 100 time steps) were indexed. Every node in the R-tree index structures was assumed to occupy one disk page. For queries covering 50% of the total time and 20% of the total space, the MON-tree required about 1/2 of the I/Os (disk accesses) required by the FNR-tree. When the query covers a smaller time interval (e.g. less than 10% of the entire time), the FNR-tree requires about 1.5 times the I/Os required by the MON-tree for the same queries.

The MultiVersion R-tree (MVR-tree) of Hadjieleftheriou et al [13] uses multiple 3-dimensional R-trees. The root node of each R-tree corresponds to a disjoint time interval. MVR-trees have time and two orthogonal spatial dimensions as the three bounding box axes. Deciding how to split moving object trajectories to balance the two opposing needs of minimizing bounding box volume while keeping the number of MVR-tree nodes to a manageable size is a signficant challenge for the MVR-tree indexing method.

Polynomial approximation (PA) trees [15] index historical moving object trajectories. The time domain $[0, T]$ is divided into m equal time intervals. Each moving object's path $f_x(t)$, $f_y(t)$ in the Cartesian x, y plane is modelled by $2m$ Chebyshev polynomials, $\hat{f}_x(t)$ and $\hat{f}_y(t)$, 2 for each time interval. Leaf nodes of the PA-tree have a 6-tuple containing the first two coefficients of the Chebyshev polynomial for each of $\hat{f}_x(t)$ and $\hat{f}_y(t)$, plus the maximum deviation errors $\epsilon_x^1 = \max_t(|\hat{f}_x(t) - f_x(t)|)$ and $\epsilon_y^1 = \max_t(|\hat{f}_y(t) - f_y(t)|)$. Leaf nodes also have a pointer to a more precise representation of the polynomial approximation that can include more than two coefficients depending on an error threshold. Interior nodes are formed similar to R*-tree nodes, except the bounding box is formed from the minimum and maximum values of the four coefficients of each child. Thus, the PA-tree has the structure of a 4-dimensional R*-tree. An interior node I contains up to M 10-tuples and M child pointers, where M is the maximum number of children for an R*-tree node. Each 10-tuple contains the minimum and maximum of the four Chebyshev polynomial coefficients stored in it's associated subtree, along with the maximum deviation errors ϵ_x^1, ϵ_y^1 in the subtree.

Ni and Ravishankar [15] illustrate why the PA-tree indexes less empty space compared to indexes using orthogonal axis-aligned bounding boxes. On a 5,000 trajectory test dataset with 1,000 timesteps (6,390,000 location updates), the PA-tree covers two to five times less empty space compared to minimum bounding rectangles covering the equivalent trajectory time intervals. This observation carries through into the experimental validation, with the PA-tree requiring about 5.5 times fewer I/Os to answer timestamp queries compared to the MVR-tree, which is based on minimum bounding rectangles. Further comparison to SETI with 1% of R type Q_1 queries, SETI required 30% fewer I/Os compared to the PA-tree index. For type Q_2 queries when $[t_1, t_2]$ spans between 5% and 10% of the time domain $[0, T]$ for which data is indexed, the PA-tree requires around 1/2 of the I/Os that SETI requires. Ni and Ravishankar [15] also arrive at a cost model for predicting the expected number of I/Os required for a PA-tree search with a given query. The model assumes a uniform random distribution of R in the unit square and $[t_1, t_2]$ in $[0, T]$, as well as a random distribution of moving object trajectories in the plane. For the experiments they ran, the I/O cost model has a relative error of no more than 25% compared to the actual number of I/Os incurred.

5 The Primary Data Structure

We assume that N objects are constrained to move on a planar graph G with E edges and V vertices. The graph can be disconnected, corresponding, for example, to disconnected road networks in a jurisdiction with islands or remote areas (as in Figure 1). The moving objects are entered m times into the data structure, giving a total of $n = mN$ location updates. The update interval UI separating update times is assumed to be constant.

The primary data structure used for indexing the planar graph is called the graph strip tree, or GStree for short. The GStree is based on the strip tree [5], but it is genaralized to allow for indexing collections of strip trees representing a planar graph. Figure 2 illustrates a planar graph with $V = E = 4$. Figure 3 illustrates the corresponding GStree arising from the planar graph in Figure 2.

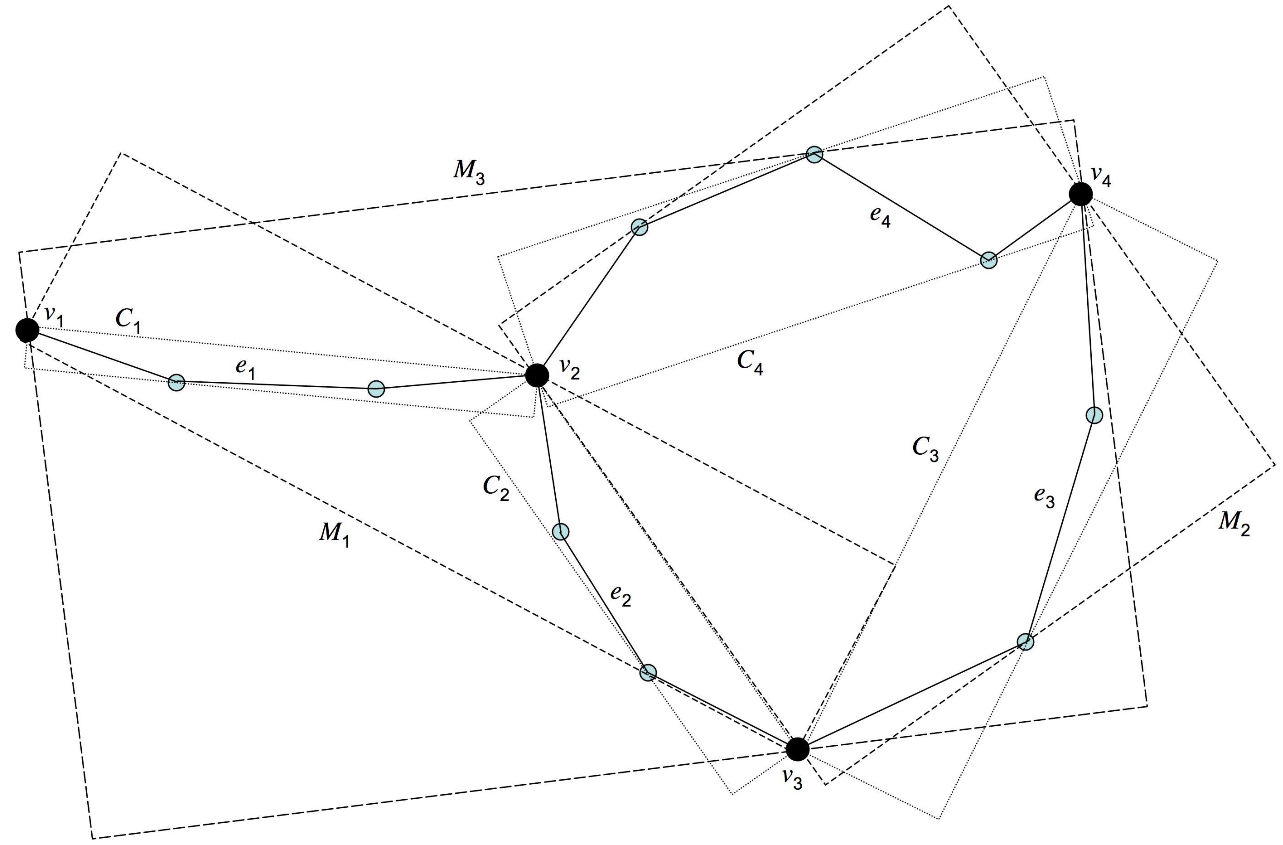

Figure 2: An example graph G with 4 edges $e_1, ..., e_4$ and 4 vertices $v_1, ..., v_4$. The edges are represented as strip trees, with $C_1, ..., C_4$ representing the root bounding boxes for each strip tree. The strip trees are merged bottom up in pairs to construct the GStree.

The static part of the GStree shown in Figure 3(a) is a binary tree as each interior node M_i has at most two children. The tree is constructed such that interior nodes have one or two children, and such that the tree is height balanced. This tree is a spatial index to the static part of the data; i.e. the planar graph on which objects move. For purposes of analysis, we assume the static part of the GStree that depends only on the underlying planar graph fits in main memory. The interval trees T_i^j indexing the dynamic part of the data are assumed to reside in external memory.

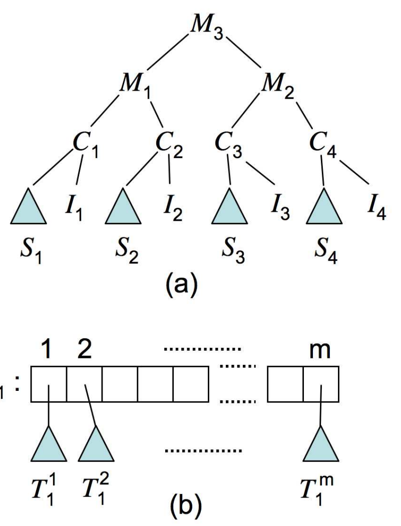

Figure 3: The graph strip tree (GStree) corresponding to the planar graph in Figure 2. (a) Each leaf C_i points to the strip tree S_i representing the graph edge e_i, as well as to the corresponding moving object interval sets I_i. (b) Interval sets I_i are an array of size m. Each element of I_i points to an external memory interval tree T_i^j representing the moving objects during interval j on edge e_i.

Theorem 1. *Assuming a constant size graph containing E edges, and containing N moving objects over m time intervals, the space required for the GStree is $O(mE)$ memory cells and $O(n/B)$ disk blocks, for B the number of elements transmitted by one external memory access.*

Proof. There are E nodes C_i, each requiring constant space. Each internal node M_i requires constant space, and there are $O(E)$ of them, so nodes M_i require $O(E)$ space. Each strip tree S_i is a balanced binary tree, with the number of leaves depending on the resolution of the strip tree, the degree of curvature of the underlying polyline comprising the graph edge, and the number of line segments making up the polyline. We assume here that there are a constant number of leaves in each strip tree, which means that all strip trees $S_i, i = 1, ..., E$ require $O(E)$ space.

Each moving object interval set I_i is an array of m pointers to m external interval trees [4]. There are E of these interval trees, one per planar graph edge, so the space required for all $I_i, i = 1, ..., E$ moving object interval sets is cmE bytes, for $c =$ the number of bytes per pointer.

Each I_i stores m pointers, each pointing to an external interval tree. Each interval tree stores some fraction of the moving objects. For time interval $[t_{j-1}, t_j]$, all N moving objects are distributed across the E external interval trees. Assuming the N moving objects are distributed in a uniform random fashion across the E edges, on average each interval tree stores N/E moving objects for each time interval. An external interval tree uses $O(g/B)$ disk blocks to store g intervals (from Theorem 4.1 of [4]). For the GSTree, $g = N/E$, and there are E external interval trees. Thus, each time interval requires $O(N/B)$ disk blocks, and m time intervals requires $O(n/B)$ disk blocks. $\square$

6 Secondary Data Structure

Each moving object interval set I_i contains m pointers to external interval trees. Each external interval tree T_i^j stores the intervals of moving objects on graph edge e_i for time interval $[t_{j-1}, t_j]$. Figure 4 illustrates a moving object interval set for edge e_1 in Figure 2.

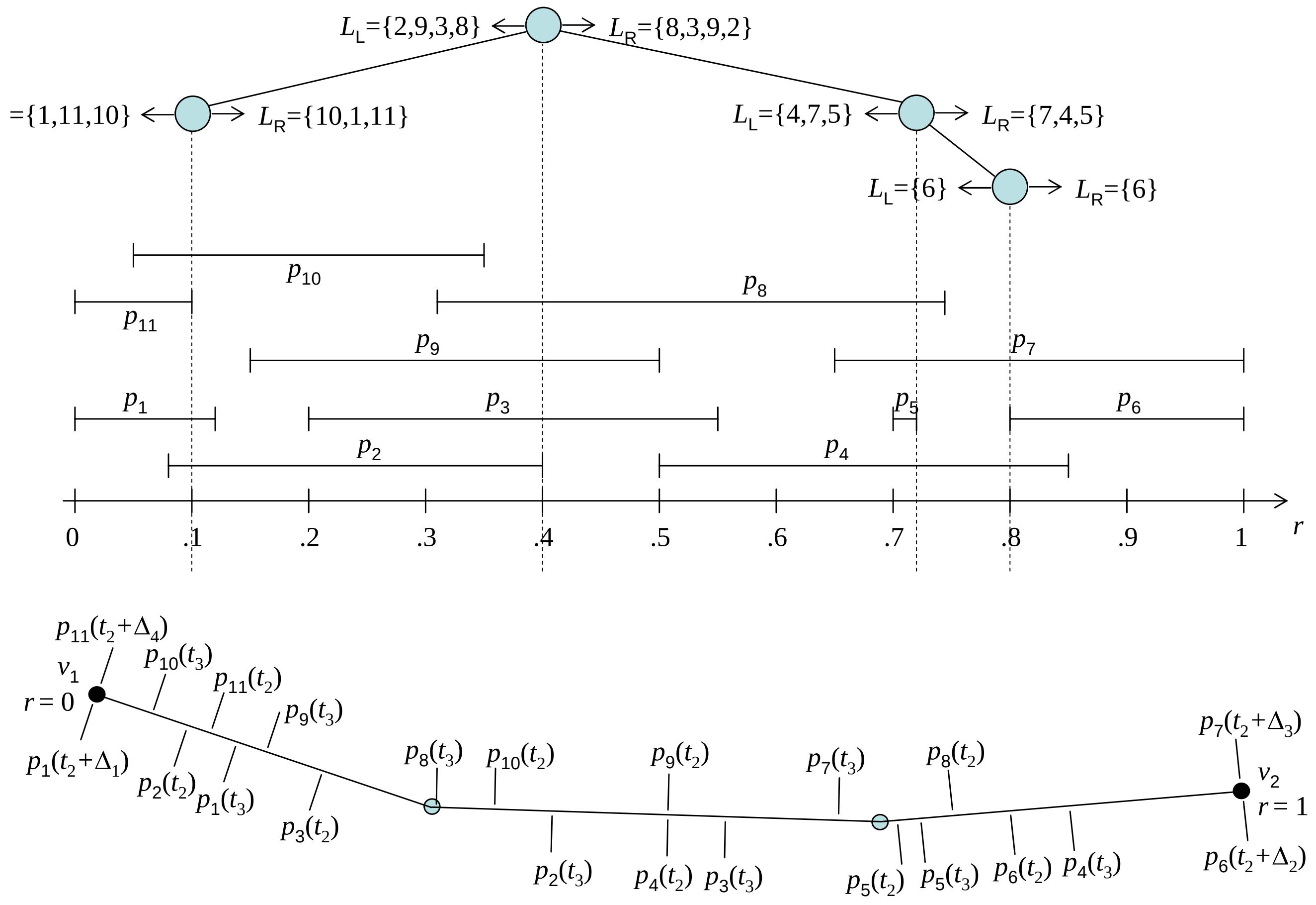

Figure 4: An example interval tree T_1^3 for e_1 of the graph in Figure 2. There are 11 intervals representing 11 moving objects during the time interval $[t_2, t_3]$. Moving objects $p_1, ..., p_6$ are moving from vertex v_1 to vertex v_2. Moving objects $p_7, ...p_{11}$ are moving in the opposite direction from v_2 to v_1.

Figure 4 also illustrates the interval tree T_1^3 arising from the interval set I_1. Interval trees are height balanced linear space data structure that can report all intervals in a set I of size g intersecting a query point in $O(\log g + K)$ time, where K is the number of reported intervals (see e.g. [10]). Interval tree T_1^3 has 4 nodes, and is height balanced.

7 Building a GStree

There are E edges e_i (i.e., polylines representing the centerlines of roads) in a planar graph respresenting a road network. We first build E strip trees S_i. These E strip trees are then merged, two at a time, to produce a packed GStree structure as shown in Figure 3.

7.1 Pack algorithm for GStree

The $PACK(Edges)$ algorithm shown in Algorithm 1 is used to pack edges (strip trees) together in a binary GStree. This algorithm works recursively and returns a pointer to the root node of a fully-packed GStree containing all strips (or edges) in Edges. Edges is a list of strips indexing edges (or road segments) of a road network.

When the number of edges in Edges is 1 or 2 (statements 2, 3, and 4), the algorithm will return a pointer to this only edges, or to a GSnode created by merging two strips in Edges. If the number of edges in Edges is more than 2 (statements 6,..., 15), the algorithm will repeatedly find the edge, E2, among edges in Edges, which has the minimum coverage with the first edge, E1, in Edges. This is done by the MinCover algorithm (Algorithm 2). E1 and E2 will be removed from Edges, merged, and pushed back to a new edge list, called MEdges, until Edges is empty. Therefore, after the while loop (statements 7, ..., 14), we obtain MEdges which is a list of packed edges. In the statement 15, the algorithm is recursively called with the new input MEdges until the packed items in MEdges is smaller than or equal to 2.

Algorithm 1: GSnode: **Pack(**$Edges, dcel$**)**

The algorithm for packing all strips

```
input  : Edges: a list of strips, indexing edges (or road segments) of a road network; dcel: a graph
         of DCEL
output: A pointer to the root node of a fully-packed GStree containing all strips (or edges) in
         Edges
1  begin
2      if $|Edges| \leq 2$ then
3          $G0 \leftarrow$ a new GSnode pointing to edges in Edges
4          return G0
5      else
6          $MEdges \leftarrow NULL$
7          while $|Edges| > 1$ do
8              $E1 \leftarrow$ the first edge of Edges
9              $Edges \leftarrow Edges - E1$
10             $E2 \leftarrow MinCover(Edges, E1, dcel)$
11             $Edges \leftarrow Edges - E2$
12             $G1 \leftarrow$ a new GSnode created by merging E1 and E2
13             $MEdges \leftarrow MEdges \cup G1$
14             if $|Edges| = 1$ then $MEdges \leftarrow MEdges \cup Edges$
15         return Pack(MEdges)
16 end
```

Such an approach leads to a minimum area coverage of all pairwise-merged strips, and thus to a minimum coverage for all oriented rectangles at all interior nodes of the GStree.

Algorithm 2: **Mincover**($GSlist, S, dcel$)

The algorithm finding a strip, which having minimal coverage when merged with an input strip

```
input  : GSlist: a list of strips; S: a strip; and dcel: a graph of DCEL
output: A strip which forms a minimal coverage (area) when merged with S
1  begin
2      $minArea \leftarrow \infty$
3      $chosenS \leftarrow NULL$
4      foreach $X$ in $GSlist$ do
5          $T \leftarrow mergeStrips(S, X)$
6          $area \leftarrow Area(T)$
7          if $area < minArea$ then
8              $minArea \leftarrow area$
9              $chosenS \leftarrow X$
10         delete $T$
11     return chosenS;
12 end
```

Algorithm 3: **mergeStrips**($S1, S2$)

The algorithm of merging two strips

```
input  : Two strips $S1$ and $S2$
output: A regular and merged strip
1  begin
2      $chosenS \leftarrow NULL$
3      $T1 \leftarrow$ strip from begin ($b$) and end ($e$) points of $S1$ and updated $wr$ and $wl$ from points of $S2$
4      $area1 \leftarrow Area(T1)$
5      if IsRegular(T1) then
6          $chosenS \leftarrow T1$
7          $minArea \leftarrow area1$
8      $T2 \leftarrow$ strip from $b$, $e$ of $S2$ and updated $wr$ and $wl$ from points of $S1$
9      $area2 \leftarrow Area(T2)$
10     if $IsRegular(T2)$ then
11         if $(chosenS = NULL)$ or $(chosenS \neq NULL$ and $area2 < area1)$ then
12             $chosenS \leftarrow T2$
13             $minArea \leftarrow area2$
14     $T3 \leftarrow$ strip from $b$, $e$ of both $S1$ and $S2$
15     $area3 \leftarrow Area(T3)$
16     if $IsRegular(T3)$ then
17         if $(chosenS = NULL)$ or $(chosenS \neq NULL$ and $area3 < minArea)$ then
           $chosenS \leftarrow T3$
18     else
19         $chosenS \leftarrow T3$ after being regularized
20     return chosenS;
21 end
```

7.2 Construct a List of Interval Trees

A moving object $p_i[t_j, t_{j+1}]$ is specified by a time interval $j = [t_j, t_{j+1}]$ and an edge e_k (road segment) on which it moves. One moving object can be stored many times depending on the number of time steps.

The update interval $UI = t_{j+1} - t_j$ is assumed to be constant. The interval tree list construction algorithm (Algorithm 4) takes as input a list of moving objects, the edge (or road) they move on during a specific time interval, and the position interval $[r_j, r_{j+1}]$ corresponding to this time interval. The output is a list of interval trees as illustrated in Figure 3.

Algorithm 4: **itreeList**($R, numObjects$)

The algorithm of creating a list of interval trees for moving objects on a road

```
input : An array $R$ contains $numObjects$ entries of moving objects. Each $R[i]$ includes the
        identify of a road $roadid$, the identify of a moving object $objectid$, a time interval $t_1$, $t_2$,
        and a position interval $r_1$, $r_2$
output: A List $I[i]$ of interval trees
1  begin
2      $R \leftarrow$ sorted in ascending order of $roadid$, then $t_1$, then $t_2$
3      $I[i] \leftarrow \emptyset$
4      $i \leftarrow 0$
5      while $i < numObjects$ do
6          $t_1 \leftarrow R[i].t_1$
7          $t_2 \leftarrow R[i].t_2$
8          $numIndex \leftarrow$ number of moving objects at time interval $[t_1, t_2]$
9          $MyInterval * ii \leftarrow$ new $MyInterval[numIndex]$
10         $ValList < double > endpoints$
11         for $j \leftarrow 0$ to $numIndex$ do
12             $ii \leftarrow insert(ii, R[j].r_1, R[j].r_2)$
13             if $notBelong(R[i].r_1, endpoints)$ then $insert(endpoints, R[i].r_1)$
14             if $notBelong(R[i].r_2, endpoints)$ then $insert(endpoints, R[i].r_2)$
15             $i \leftarrow i + 1$
16         $sort(endpoints)$
17         $IntervalTree < MyInterval > itree(endpoints)$
18         $itree.t_1 = t_1$
19         $itree.t_2 = t_2$
20         for $k \leftarrow 0$ to $numIndex$ do $insert(itree, ii[k])$
21         Insert $itree$ into $I[i]$
22     return $I[i]$
23 end
```

8 Searching in a GStree

A search with query $Q_2 = (R, [t_1, t_2])$, in the form of a rectangular query R and a time interval $[t_1, t_2]$, is performed from the root node to the leaf nodes of the GStree. The rectangular query R

is used first to find edges intersecting R. For each intersected edge, the time interval query $[t_1, t_2]$ is used in interval trees to find moving objects satisfying the query.

The GStree is first used to find position intervals in each edge intersecting the query rectangle R. Figure 5 illustrates three cases that can arise.

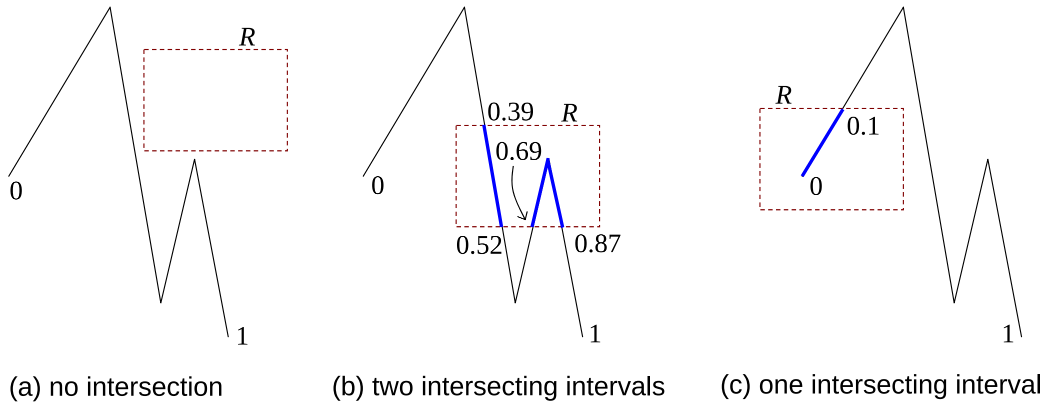

Figure 5: Three cases for the intersection between a graph edge and a query rectangle R.

If an edge e_i does intersect R, a list F_i of intersecting intervals between e_i and R is constructed. For example, there are two intersecting intervals $[0.39, 0.52]$ and $[0.69, 0.87]$ in Figure 5 (b), and there is one intersecting interval $[0, 0.1]$ in Figure 5 (c). Interval trees whose time interval intersects with the query time interval $[t_1, t_2]$ are used for the next search step.

With the input lists F_i, the algorithm searches in the appropriate interval trees T_i^j and returns the list D_i of moving objects intersecting the query intervals. The appropriate interval trees T_i^j to be searched are those for which $t_a \leq t_1$ and $t_b \geq t_2$, for $a, b \in 1, ..., m$. This guarantees that all retrieved intervals representing moving objects are within UI of the query time t_q (for Q_1 queries) or query interval $[t_1, t_2]$ (for Q_2 queries).

Algorithms 5 and 6 show the search algorithm for Q_1 type queries in the GStree. Algorithms 8 and 9 show the search algorithm for Q_2 type queries in the GStree. The searchObject algorithm (see Algorithm 7) performs a range search to find all intervals in the interval tree intersecting the query fragment range $[r_1, r_2]$. In the theorem below, we assume that the GSTree 'upper part' (i.e. all except the external interval trees T_i^j) can be stored in main memory. Thus, no I/Os are required to search precisely which strip trees S_i are intersected, and where.

Theorem 2. *For N moving objects randomly distributed on the E edges of a planar graph, the number of I/Os required to determine the moving objects intersecting one edge at time t_q in a GStree is expected to be $O(\log_B \frac{N}{E} + k)$, where k is the number of disk blocks required to store the answer.*

Proof. We assume that the number of fragments $|F_i|$ for one edge e_i intersecting R is constant. Algorithm **searchObject** performs two time instant queries (one for r_1 and one for r_2), and collects all intervals intersecting $[r_1, r_2]$ by visiting all interior nodes in the subtree rooted at the nearest common ancestor of the leaves visited for the r_1 and r_2 time instant queries. From Theorem 4.1 of Arge and Vitter [4], we know that an external interval tree containing g intervals can answer a time instant query in $O(\log_B g + k)$ I/Os in the worst case. We have m external interval trees for each edge e_i, only one of which is visited for $Q_1 = (R, t_q)$ queries. For randomly distributed moving objects, we expect N/E moving objects in each external interval tree, giving an expected number N/E intervals in one external interval tree T_i^j. The two time instant queries thus require $O(\log_B \frac{N}{B} + k)$ I/Os. Internal nodes visited on the path from the nearest common ancestor contribute intervals falling between r_1 and r_2. The disk I/Os needed to retrieve internal node intervals are counted towards the $O(k)$ disk blocks needed to store the answer. □

Given a GStree tree built from a planar graph G containing E edges, and containing N moving objects on m time intevals, the above theorem shows that the expected number of I/Os required to answer a Q_1 query is $O(JE \log_B \frac{N}{E} + k)$, where k is the number of disk blocks required to store the answer, and J is the fraction $\in [0, 1]$ of the edges in planar graph G intersected by R.

Algorithm 5: **GSSearch(**M_{root}, R, t_q**)**

The algorithm for Q1 searching on the GStree.

```
input  : Root node of GStree: $M_{root}$, query rectangle $R$, and query time $t_q$
output: List of moving objects satisfying $Q_1 = (R, t_q)$
1  begin
2      if $intersect(M_{root}, R)$ then
3          if $M_{root}.attribute = C_i$ then
4              $fraglist \leftarrow FragIntervals(M_{root}, M_{root}.id, R)$
5              if $fraglist \neq NULL$ then
6                  return $intervalSearch(I_i, t_q, fraglist)$
7          else
8              return $GSSearch(M_{root}.left, R, t_q) \cup GSSearch(M_{root}.right, R, t_q)$
9      else
10         return $\emptyset$
11 end
```

Algorithm 6: **intervalSearch(**$I_{root}, t_q, fraglist$**)**

The algorithm for Q1 searching on a list of interval trees I_{root}

```
input  : itree list: $I_{root}$, query time $t_q$, and list of query interval fragments: $fraglist$
output: List of unique moving objects in $I_{root}$ satisfying $t_q$ and intersecting at least one fragment
        in $fraglist$
1  begin
2      $oList \leftarrow \emptyset$
3      $T_{root}^j \leftarrow$ interval tree, $\in I_{root}$ containing time $t_q$, found using a binary search
4      while $fraglist \neq NULL$ do
5          push($oList, searchObject(T_{root}^j, fraglist.r_1, fraglist.r_2)$)
6          $fraglist \leftarrow fraglist.next$
7      return $oList$
8  end
```

Algorithm 7: **searchObject**($itree, r_1, r_2$)

The algorithm for searching on an interval tree itree

```
input  : Root node of interval tree $itree$, and positional interval $[r_1, r_2]$
output: List of moving objects in $itree$ intersecting with $[r_1, r_2]$
1  begin
2      $oList \leftarrow \emptyset$
3      if $itree = NULL$ then return $\emptyset$
4      if ($itree.split \geq r_1$) and ($itree.split \leq r_2$) then
5          if $itree.minlist \neq NULL$ then
6              while $tmp \leftarrow itree.minlist.next$ do
7                  push($oList, tmp.oid$)
8          push($oList, searchObject(itree.left, r_1, r_2) \cup searchObject(itree.right, r_1, r_2)$)
9      else
10         if $itree.split > r_2$ then
11             if $itree.minlist \neq NULL$ then
12                 while ($tmp \leftarrow itree.minlist.next$) and ($tmp.min \leq r_2$) do
13                     push($oList, tmp.oid$)
14             push($oList, searchObject(itree.left, r_1, r_2)$)
15         else
16             if $itree.split < r_1$ then
17                 if $itree.maxlist \neq NULL$ then
18                     while ($tmp \leftarrow itree.maxlist.next$) and ($tmp.max \geq r_1$) do
19                         push($oList, tmp.oid$)
20                 push($oList, searchObject(itree.right, r_1, r_2)$)
21     return $oList$
22 end
```

Algorithm 8: **GSSearch**(M_{root}, R, t_1, t_2)

The algorithm for Q2 searching on the GStree.

```
input  : Root node of GStree: $M_{root}$, query rectangle $R$, and query time interval $[t_1, t_2]$
output: List of moving objects satisfying $Q_2 = (R, [t_1, t_2])$

1  begin
2      if $intersect(M_{root}, R)$ then
3          if $M_{root}.attribute = C_i$ then
4              $fraglist \leftarrow FragIntervals(M_{root}, M_{root}.id, R)$
5              if $fraglist \neq NULL$ then
6                  return $intervalSearch(I_i, t_1, t_2, fraglist)$
7          else
8              return $GSSearch(M_{root}.left, R, t_1, t_2) \cup GSSearch(M_{root}.right, R, t_1, t_2)$
9      else
10         return $\emptyset$
11 end
```

Algorithm 9: **intervalSearch**($I_{root}, t_1, t_2, fraglist$)

The algorithm for Q2 searching on a list of interval trees I_{root}

```
input  : itree list: $I_{root}$, query time interval $[t_1, t_2]$, and list of query interval fragments: $fraglist$
output: List of unique moving objects in $I_{root}$ satisfying $[t_1, t_2]$ and intersecting at least one
        fragment in $fraglist$

1  begin
2      $oList \leftarrow \emptyset$
3      for each $T^j_{root} \in I_{root}$ do
4          if $T^j_{root}.tmin \leq t_2$ and $T^j_{root}.tmax \geq t_1$ then
5              while $fraglist \neq NULL$ do
6                  $uniqueL \leftarrow$ unique(searchObject($T^j_{root}, fraglist.r_1, fraglist.r_2$))
7                  push($oList, uniqueL$)
8                  $fraglist \leftarrow fraglist.next$
9      return $oList$
10 end
```

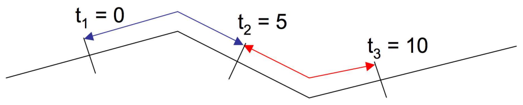

Figure 6: Example of a generated moving object with two time steps. For example, with velocity of 15 km/h (250 m/minute), at each time step the object moves 1,250 m.

9 Experimental Results

9.1 Implementation

The GStree and the MON-tree were implemented using C++. We then compare the GStree to the MON-tree and to naïve search.

Our focus is on moving objects (e.g., vehicles) on a planar graph, and we used the New Brunswick road network (see Figure 7), road data from a Canadian road network [19] for testing. This data file contains 66,437 roads (i.e., polylines), each of which is treated as an edge in our planar graph E. All test data for the MON-tree and GStree are prepared as follows:

1. We first store the 66,437 edges of the road network into a doubly connected edge list (DCEL) (e.g. [10]). The DCEL helps us to keep objects moving on the planar graph, whose edges are bidirectional. When a vehicle moves to the end of an edge, it will exit onto an edge connected to it, or turn back in the same edge with the opposite direction if the edge is unconnected.

2. We next randomly generate r_1 (normalized first position) of a moving object on edge e_i on the DCEL graph. We assume that in each edge, the velocity y_i of moving objects is the same. In addition, velocities are randomly generated between a maximum velocity y_{max} (e.g., 100 km/h) and a minimum velocity y_{min} (e.g., 10 km/h).

3. For each time step, compute r_2 using the edge velocity y_i, the beginning normalized position r_1 and the update interval UI (e.g. 5 minutes). Repeat for all m time steps, each time computing the normalized position interval (r_1, r_2) by setting r_1 to the previous r_2 value while accounting for "end of edge" conditions.

Algorithm 10 $Generator(E, m, UI)$ is used to generate random data of moving objects on E roads with m time steps. Each time step (or time interval) is UI minutes . The randomly generated data is kept in text files as input for testing the MON-tree, the GStree, and the naïve search. In Algorithm 10, $insert(o_j, e_i, r_0, r_1, t_0, t_1)$ is used to insert a moving object o_j on the road e_i with its position interval (r_0, r_1) and its time interval (t_0, t_1) into a text file. The numbers of moving objects on an edge is constrained to lie between 4 and 40 per km of edge length. For the test graph shown in Figure 7, this resulted in a total of 443,983 moving objects in the test data. For $m = 5$, 3,288,689 total instances of moving objects arose as objects move from one edge to another when arriving at a vertex v in the graph.

Algorithm 10: **Generator**(E, m, UI)

The algorithm for generating random data of moving objects

```
Input  : $E$ roads with $m$ time steps, each of which contains $UI$ minutes.
output: Random data of moving objects on $E$ roads with $m$ time steps.
1  begin
2      for $i = 0$ to $E - 1$ do $velo_i \leftarrow$ a random velocity $\in \{V_{min}, V_{max}\}$
3      for $i = 0$ to $E - 1$ do
4          $numobject \leftarrow$ a random number $\in \{mino(e_i), .., maxo(e_i)\}$ of moving objects on road $e_i$
5          if $numobject \neq 0$ then
6              foreach $o_j \in \{o_0, .., o_{numobject}\}$ do
7                  $r_1 \leftarrow$ the first random position for $o_j$
8                  $t_1 \leftarrow 0$; $velo \leftarrow velo_i$
9                  $direction \leftarrow$ current direction of $e_i$
10                 for $step \leftarrow 1$ to $m$ do
11                     $t_0 \leftarrow t_1$; $t_1 \leftarrow t_0 + UI$
12                     $r_0 \leftarrow r_1$; $r_1 \leftarrow r_0 + velo \times UI$
13                     if $overRoad(r_1)$ then
14                         $t_{mid} \leftarrow$ time when current object at the end of $e_i$
15                         $r_1 \leftarrow$ position at the end point of $e_i$
16                         insert($o_j, e_i, r_0, r_1, t_0, t_{mid}$)
17                         $t_0 \leftarrow t_{mid}$
18                         if $incidents(e_i) > 0$ then
19                             $e_k \leftarrow$ a random incident road of $e_i$
20                             $e_i \leftarrow e_k$; $velo \leftarrow velo_k$
21                             $r_0 \leftarrow 0$; $r_1 \leftarrow velo \times (t_1 - t_0)$
22                             $direction \leftarrow$ current direction of $e_k$
23                         else
24                             $direction \leftarrow$ reverse current direction
25                             $r_0 \leftarrow$ position at the end point of $e_i$
26                             $r_1 \leftarrow$ position after $t_{mid}$ time moving of current object
27                     insert($o_j, e_i, r_0, r_1, t_0, t_1$)
28                     if $endRoad(r_1)$ then
29                         if $incidents(e_i) > 0$ then
30                             $e_k \leftarrow$ a random incident road of $e_i$
31                             $e_i \leftarrow e_k$; $v \leftarrow v_k$
32                             $direction \leftarrow$ current direction of $e_k$
33                             $r_1 \leftarrow$position at the start point of $e_k$
34                         else
35                             $direction \leftarrow$ reverse current direction
36                             $r_1 \leftarrow$ position at the end point of $e_i$
37 end
```

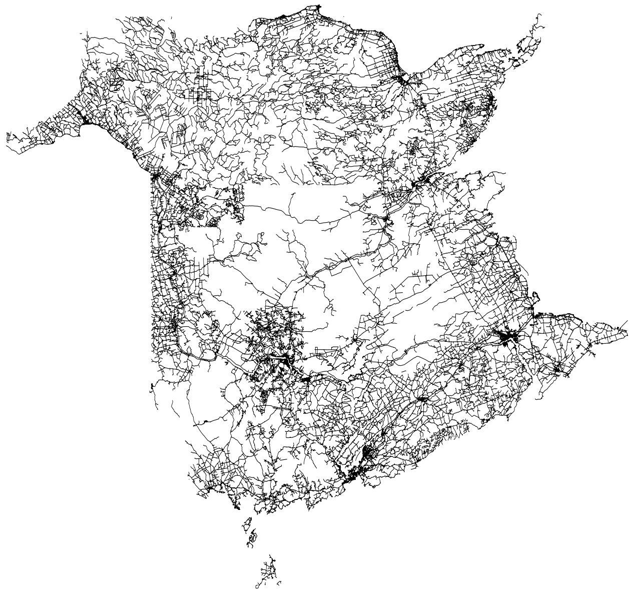

Figure 7: The road network of New Brunswick consisting of a planar graph with 66,437 edges and 54,827 vertices (ratio of 1.21). The average, minimum and maximum length (in m) of polylines defining the edges is 694, 5 and 31,334, respectively. The average number of points in a polyline is 7. This picture was drawn using the TatukGIS Viewer open source tool [1].

9.2 Comparing the GStree to the MON-tree and naïve search

The tree building algorithm and search algorithms for the GStree, the MON-tree, and naïve search are run on a Linux-based parallel cluster with 62 nodes called "mahone2". Each node has 16 GB RAM and two AMD Opteron 2.6 (or 2.8) GHz processors. All experiments were run on a single processor.

9.2.1 Time for searching

We ran both trees and naïve search with the same set of 443,983 moving objects on 66,437 roads. One set of 400 random queries was generated, and the same set was used for each test. In our testing, query rectangles R are from 1 to 10 percent of the size of the road network bounding box. A random query is generated in three steps. First, we randomly generated the central point of the query, which must fall inside the road network bounding box. Second, we generated random vertical and horizontal sizes. Finally, a time $t_q \in [0, T]$ or time interval $[t_1, t_2]$, for $t_1 \in [0, T]$ and $t_1 < t_2$, for the query was also randomly generated.

We categorized queries into five types based on the number K of moving objects meeting the query requirements. The five query result ranges are [0, $log_2^{1/2}(n)$), [$log_2^{1/2}(n)$, $log_2(n)$), [$log_2(n)$, $log_2^2(n)$), [$log_2^2(n)$, $log_2^3(n)$), and [$log_2^3(n)$, n], called query range 1, 2, 3, 4, and 5, respectively. Note that $n = mN$ is the total number of instances of moving objects in a tree. When n = 3,288,689, then the upper bounds for ranges 1 though 4 are 5, 22, 469 and 10,147, respectively. Table 2 shows an example of the number of objects falling in these categorized queries. Algorithm 11 shows the test harness code for *numQuery* queries.

Algorithm 11: **GlobalSearching**($M_{root}, Q, numQuery, n$)

The test harness code for numQuery queries.

input : Root of the GStree M_{root}, a set of *numQuery* queries, n moving objects

output: Show average search time and average visited nodes for five query ranges.

```
begin
    $d_1, d_2, d_3, d_4, d_5 \leftarrow 0$  //number of visited nodes for five query ranges of queries
    $h_1, h_3, h_3, h_4, h_5 \leftarrow 0$  //number of queries
    $time_{h_1}, time_{h_2}, time_{h_3}, time_{h_4}, time_{h_5} \leftarrow 0$  //total time for executing queries
    for $i = 1$ to numQuery do
        $R, t_1, t_2 \leftarrow$ query rectangle and query time interval of $Q_i$
        $olist \leftarrow GSSearch(M_{root}, R, t_1, t_2)$
        $F \leftarrow oList.size()$  //number of objects in range
        $time_{total} \leftarrow$ search time of this query
        $D \leftarrow$ number of visited nodes of this query
        switch the value of $F$ do
            case $[0, log_2^{1/2} n)$
                $d_1 \leftarrow d_1 + D$
                $h_1 \leftarrow h_1 + 1$
                $time_{h_1} \leftarrow time_{h_1} + time_{total}$
            case $[log_2^{1/2} n, log_2 n)$
                $d_2 \leftarrow d_2 + D$
                $h_2 \leftarrow h_2 + 1$
                $time_{h_2} \leftarrow time_{h_2} + time_{total}$
            case $[log_2 n, log_2^2 n)$
                $d_3 \leftarrow d_3 + D$
                $h_3 \leftarrow h_3 + 1$
                $time_{h_3} \leftarrow time_{h_3} + time_{total}$
            case $[log_2^2 n, log_2^3 n)$
                $d_4 \leftarrow d_4 + D$
                $h_4 \leftarrow h_4 + 1$
                $time_{h_4} \leftarrow time_{h_4} + time_{total}$
            otherwise
                $d_5 \leftarrow d_5 + D$
                $h_5 \leftarrow h_5 + 1$
                $time_{h_5} \leftarrow time_{h_5} + time_{total}$
    return {average query time, average number of visited nodes for each query range}
    $\{time_{h_1} \div h_1, d_1 \div h_1\}$
    $\{time_{h_2} \div h_2, d_2 \div h_2\}$
    $\{time_{h_3} \div h_3, d_3 \div h_3\}$
    $\{time_{h_4} \div h_4, d_4 \div h_4\}$
    $\{time_{h_5} \div h_5, d_5 \div h_5\}$
end
```

Table 2: Example query result categorization ranges. n is the number of entries of moving objects.

Range	1	2	3	4	5
n	$[log_2^{1/2}n]$	$[log_2 n]$	$[log_2^2 n]$	$[log_2^3 n]$	$> [log_2^3 n]$
3,288,689	4	21	468	10,146	

To benchmark the search times of the algorithms, we ran 400 random queries on a set of the generated data for n = 3,288,689 entries (443,983 moving objects with 5 time steps). We counted the average search time and the number of visited nodes D. For the MON-tree D is the number of visited nodes in the bottom R*-trees. For the GStree D is the total number of nodes visited in interval trees. As for the experiments done by de Almeida and Güting [9], the MON-tree was constructed with R*-trees having nodes with a maximum number of children M that fit on one disk block. In our case, the disk block size is 4,096 bytes, 56 bytes are required for the bounding box and pointer information for one child of the R*-tree, so $M = 73$. Naïve search defines an array of size n, and scans all n elements of the array to determine those moving objects matching the query.

Table 3 shows the Q_1 search results for the five query ranges 1, 2, 3, 4 and 5, respectively on the GStree, the MON-tree, and naïve search. Note that numbers in parenthesis are the ratios of the current number of the MON-tree or naïve search to the corresponding number of the GStree. Table 4 shows the Q_2 search results for the same five query ranges. Similarly, Tables 5 and 6 shows search results of Q_1 and Q_2, respectively, when $m = 50$. Table 7 shows search results for the same five query ranges, but for the GStree and naïve search only with $m = 100$. At the time of submission of this technical report, the MON-tree ran out of memory when attempting to run experiments for $m = 100$.

Table 3: Average search times (in seconds) and numbers of visited nodes for the GStree, the MON-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. D is the average number of visited nodes.

Range	h	GStree		MON-tree		naïve
		time	D	time	D	time
1	179	0.00043	0.03	0.00009(0.22)	1.28(38.17)	0.123(284.96)
2	3	0.00233	7.00	0.00500(2.14)	100.67(14.38)	0.125(53.57)
3	32	0.00355	167.52	0.01187(3.35)	832.44(4.97)	0.123(34.79)
4	180	0.01941	1,659.21	0.25942(13.37)	10,595.50(6.39)	0.132(6.78)
5	6	0.07749	7,276.33	1.87738(24.23)	45,371.00(6.24)	0.160(2.07)

10 Discussion

For low numbers of points in range, the in-memory search experiments performed here show that the MON-tree is up to 5 times faster, but with significantly more nodes accessed (on average) compared to the GStree. If one disk access is required for one visited node of the MON-tree, then the number of I/Os required for searching in a MON-tree will likely be much higher than

Table 4: Average search times (in seconds) and numbers of visited nodes for the GStree, the MON-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. D is the average number of visited nodes.

Range	h	GStree		MON-tree		naïve
		time	D	time	D	time
1	179	0.00047	0.03	0.00012(0.26)	1.28(38.17)	0.122(257.61)
2	2	0.00300	9.50	0.00150(0.50)	81.00(8.53)	0.124(41.50)
3	23	0.00309	195.00	0.00804(2.61)	646.39(3.31)	0.123(40.00)
4	173	0.01722	2,287.62	0.15290(8.88)	9,249.58(4.04)	0.133(7.73)
5	23	0.05921	10,039.35	0.74206(12.53)	36,595.87(3.65)	0.165(2.79)

Table 5: Results of Q1 when m=50. Average search times (in seconds) and numbers of visited nodes for the GStree, the MON-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. D is the average number of visited nodes.

Range	h	GStree		MON-tree		naïve
		time	D	time	D	time
1	157	0.00052	0.02	0.00012(0.23)	1.35(70.00)	1.120(2156.47)
2	4	0.00160	18.00	0.00500(3.13)	213.40(11.86)	1.112(694.83)
3	57	0.00598	235.74	0.14524(24.29)	2240.54(9.50)	1.123(187.78)
4	180	0.03400	2028.48	1.30556(38.40)	25543.09(12.59)	1.130(33.22)
5	2	0.19347	12362.50	4.81927(24.91)	189054.00(15.29)	1.254(6.48)

Table 6: Results of Q2 when m=50. Average search times (in seconds) and numbers of visited nodes for the GStree, the MON-tree, and the naïve search. h is the number of range searches (out of 400) that were averaged to obtain these results. D is the average number of visited nodes.

Range	h	GStree		MON-tree		naïve
		time	D	time	D	time
1	155	0.00057	0.00	0.00016(0.28)	1.00	1.120(1973.45)
2	5	0.00200	14.75	0.00325(1.63)	167.25(11.34)	1.116(558.08)
3	39	0.00491	237.24	0.03348(6.82)	1496.36(6.31)	1.137(231.65)
4	195	0.02811	2972.32	0.42108(14.98)	20726.65(6.97)	1.142(40.63)
5	6	0.12077	12834.64	3.04339(25.20)	116543.64(9.08)	1.196(9.91)

Table 7: Average search times (in seconds) and numbers of visited nodes for the GStree and the naïve search at 100 time steps. h is the number of range searches (out of 400) that were averaged to obtain these results. D is the average number of visited nodes.

	Range	GStree			naïve
		h	time	D	time
Q1	1	158	0.00106	0.28	2.150(2034.53)
	2	4	0.01125	11.25	2.184(194.20)
	3	61	0.06024	218.16	2.169(36.02)
	4	177	0.61456	2013.59	2.181(3.55)
Q2	1	156	0.00053	0.01	2.210(4154.55)
	2	5	0.00360	23.40	2.279(633.32)
	3	41	0.01905	253.09	2.213(116.15)
	4	195	0.24437	3227.82	2.244(9.18)
	5	3	1.05821	14189.38	2.269(2.14)

for searching in a GStree. The external memory interval tree of Arge and Vitter [4] is optimal in the worst case. The GStree is directly implementable as an I/O efficient data structure by replacing the internal memory interval trees with the optimal external memory interval tree (e.g. as implemented by Chiang and Silva [8]). Optimal I/O-efficient versions of R-trees can be used (e.g. the priority R-tree [3]) for the MON-tree. The GStree will still likely require fewer I/Os due to the top spatial index composed of oriented rectangular strips rather than axis-aligned bounding boxes used by R-trees. This conjecture is supported by the observation in Tables 3 and 4 for query range 1 (with less than $\log^{1/2} n$ moving objects in range). The GStree averages 0.03 interval tree nodes visited whereas the MON-tree averages 1.28 bottom R*-tree nodes for the same set of queries. This implies that the GStree top structure (merged strip trees indexing the planar graph) is pruning the search space more efficiently than the top R*-tree of the MON-tree.

11 Conclusion

In this paper we present a new data structure, the so-called GStree, for efficient search of moving objects (e.g., vehicles) on planar graphs. The GStree is a combination of strip trees and interval trees. Strip trees are used for indexing edges in a planar graph. Each strip tree (at leaf level) represents a polyline (corresponding to a road or road segment in a road network). The interval trees are used to index the trajectories of moving objects on roads indexed by strip trees. There are some advantages for the GStree. First, the top strip trees and the bottom interval trees are independent; thus, we can update one of them without affecting the others. For example, one can update interval trees without changing the strip tree indexing for edges, or update a strip tree when an edge changes, without affecting other strip trees (at leaf level). Second, since moving objects on a graph edge belong to a strip tree, we can easily answer queries which count moving objects on a specific edge; for example, how many vehicles move on a specific road at a specific time or during a specific time interval.

It remains to experimentally validate the GStree with an implementation of I/O-efficient external memory interval trees (e.g. [8]).

12 Acknowledgements

This research is supported, in part, by the Natural Sciences and Engineering Research Council (NSERC) of Canada, the UNB Faculty of Computer Science, the Harrison McCain Foundation, MADALGO - Center for Massive Data Algorithmics (a Center of the Danish National Research Foundation) and the government of Vietnam.

References

[1] The web page: Tatukgis viewer: Geographic information system software products and solutions. http://www.tatukgis.com/Home/home.aspx, Last accessed: June 15, 2007.

[2] P. K. Agarwal, L. Arge, and J. Erickson. Indexing moving points. *Journal of Computer and System Sciences*, 66:207–243, 2003.

[3] L. Arge, M. D. Berg, H. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Trans. Algorithms*, 4(1):1–30, 2008.

[4] L. Arge and J. S. Vitter. Optimal external memory interval management. *SIAM J. Comput.*, 32(6):1488–1508, 2003.

[5] D. H. Ballard. Strip trees: a hierarchical representation for curves. *Communications of ACM*, 24(5):310–321, 1981.

[6] J. Basch, L. J. Guibas, and J. Hershberger. Data Structures for Mobile Data. In *SODA: ACM-SIAM Symposium on Discrete Algorithms (A Conference on Theoretical and Experimental Analysis of Discrete Algorithms)*, pages 747–756, New Orleans, Louisiana, US, 5-7 January, 1997.

[7] V. P. Chakka, A. C. Everspaugh, and J. M. Patel. Indexing large trajectory data sets with seti. In *Proceedings of the Conference on Innovative Data Systems Research (CIDR 2003)*, Jan. 5-8, 2003.

[8] Y.-J. Chiang and C. T. Silva. External memory techniques for isosurface extraction in scientific visualization. In *in External Memory Algorithms and Visualization, DIMACS Series in Discrete Mathematics and Theoret. Comput. Science 50*, pages 247–277, 1999.

[9] V. T. de Almeida and R. H. Güting. Indexing the trajectories of moving objects in networks. *GeoInformatica*, 9(1):33–60, 2005.

[10] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry Algorithms and Applications*. Springer-Verlag, 2000.

[11] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *J. Comput. Syst. Sci.*, 38(1):86–124, 1989.

[12] E. Frentzos. Indexing objects moving on fixed networks. In *Proceedings of the 8th International Symposium on Spatial and Temporal Databases*, pages 289–305, Santorini, Greece, July 24-27, 2003.

[13] M. Hadjieleftheriou, G. Kollios, V. J. Tsotras, and D. Gunopulos. Indexing spatiotemporal archives. *VLDB Journal*, 15(2):143–164, 2006.

[14] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of SIGMOD 2004*, pages 623–634, 2004.

[15] J. Ni and C. V. Ravishankar. Indexing spatio-temporal trajectories with efficient polynomial approximations. *IEEE Transactions on Knowledge and Data Engineering*, 19(5):663–678, May 2007.

[16] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *VLDB 2000, Proceedings of 26th International Conference on Very Large Data Bases, September 10-14, 2000, Cairo, Egypt*, pages 395–406. Morgan Kaufmann, 2000.

[17] J. F. Roddick, M. J. Egenhofer, E. G. Hoel, D. Papadias, and B. Salzberg. Spatial, temporal and spatio-temporal databases - hot issues and directions for PhD research. *SIGMOD Record*, 33(2):126–131, June, 2004.

[18] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD Conference*, pages 331–342, Dallas, Texas, United States, May 15 - 18, 2000.

[19] Statistics and Canada. 2006 road network file. http://geodepot.statcan.ca/Diss2006/DataProducts/RNF2 last accessed: June 24, 2008.

[20] M. Vazirgiannis and O. Wolfson. A spatiotemporal model and language for moving objects on road networks. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases. Lecture Notes in Computer Science, Springer-Verlag*, volume 2121, pages 20–35, London, UK, July 12 - 15, 2001.