Optimal Space Data Structures for I/O Efficient Search of Objects Moving on a Graph

by

Thuy T. T. Le and Bradford G. Nickerson

TR09_195, October 03, 2009

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 space optimal spatio-temporal data structure called minimum I/O Graph Strip Tree (*minGStree*)to index objects assumed to move at constant velocity on the edges of a graph. The *minGStree* is designed to efficiently answer time instance and time interval queries about the past positions of moving objects. The *minGStree* uses $\Theta(\frac{n}{B})$ blocks of external memory, where n is the number of moving object instances (unique entries of moving objects) and B is the I/O block size. We propose two variations of the *minGStree*: $minGStree_I$ and $minGStree_R$. For n moving object instances distributed in a uniform fashion on the E edges of a graph over a time domain [0; T], the expected number of I/Os required to determine the moving objects intersecting one edge (for both time instance queries and time interval queries) in a $minGStree_I$ is $O(log_B(\frac{n}{E})+k)$. In the $minGStree_R$ variation, $O(\sqrt{\frac{n}{E}}+k''')$ I/Os are required for the same query. k and k''' are the number of disk blocks required to store the time intervals intersecting the time query, and the rectangles intersecting the query, respectively. We anticipate this data structure will be practical to implement.

1 Introduction

A significant challenge in spatio-temporal databases is how to improve the response time for query processing of moving objects. In these large databases, where data are stored in external memory, the response time depends on the number of I/Os (disk accesses) required to process queries. Saltenis et al [13] divide the problem of indexing the positions of continuously moving objects into two categories: indexing the current and future positions of moving objects, and indexing the historical positions of moving objects. Our research addresses the latter category. Queries on historical data are likely to be used in applications such as planning, event reconstruction and training.

Two basic approaches are used to index 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 Agarwal et al [1], so that a feasibly sized index can be built for (potentially) many moving objects. The second approach considers updates arriving at regular intervals for all objects [10]. 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.

Most previous work for indexing moving objects assumes free movement of the objects in space. If movement is restricted to edges of a graph, the index should be able to use less storage than would be required if objects were free to move anywhere in space. Our approach takes advantage of this restricted movement, and updates moving objects twice on an edge (i.e., when objects enter or leave edges).

2 Our Results

We address the problem of indexing moving objects on a graph (possibly disconnected) defined by its edges and vertices. The graph can be non-planar as it is when representing road networks [8]. We propose two variants of the *minGStree*: an I/O interval tree version $minGStree_I$ and priority R-tree version $minGStree_R$. We use *minGStree* to refer to both versions. The *minGStree* data structure extends previous work that gave rise to the GStree [11]. The *minGStree* requires $\Theta(n/B)$ disk blocks by having only one time-based I/O efficient interval tree T_i per edge, or one priority R-tree per edge. This efficient space bound assumes that $n > BE$; i.e., there are many more moving object instances than edges on which they move. This assumption is reasonable for objects moving on a static graph, especially when longer time domain $[0, T]$ are considered. 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).

For n moving object instances randomly distributed over E edges, we show that the I/Os required to determine the moving objects on an edge satisfying a query (Q_1 or Q_2) is expected to be $O(\log_B \frac{n}{E} + k)$ for the $minGStree_I$, and $O(\sqrt{\frac{n}{E}} + +k''')$ for the $minGStree_R$, where k is the number of disk blocks required to store the time intervals, and k''' is the number of disk blocks required to store the rectangles intersecting the query Q_1 or Q_2.

3 Related Work

There has been significant research into indexes able to store complete histories of data repositories. Kinetic data structures [5] 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 [1] and the references therein. A recent paper by Ni and Ravishankar [12] contains a good overview of data structures experimentally validated for moving object indexing. The characteristics of some of the known approaches for indexing moving objects are summarized in Table 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 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 *minGStree* is designed to

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 graph, L = movement constrained to be piecewise linear, E = experimental validation.

Name	H?	F?	C?	L?	E?	
TPR-tree	N	Y	N	Y	Y	[13]
partition tree	N	N	N	Y	N	[1]
kinetic range tree	N	Y	N	N	N	[1]
MON-tree	Y	N	Y	Y	Y	[7]
MVR-tree	Y	N	Y	Y	Y	[10]
PA-tree	Y	N	N	N	Y	[12]
PPFI	Y	Y	Y	Y	Y	[9]
minGStree	Y	N	Y	Y	N	this paper

exploit this constraint.

Recently, the MON-tree [7] and PPFI [9] data structures combine R-trees to index moving objects on a fixed network. In these approaches, a network is indexed first in an R-tree. Moving objects are indexed on a forest of R-trees, whose roots are linked to leaf nodes of the network tree. Our data structure utilizes strip trees [4] to index the network or graph, and interval trees (or priority R-tree) to efficiently index objects moving on the graph. While strip trees have space advantages on indexing polylines, and I/O interval trees support an optimal range search time for time intervals one edge, we believe that combining them will introduce an efficient data structures to index objects moving on edges.

To our knowledge, our research is the first to explore the theoretical performance of data structures indexing objects moving on a graph.

4 The Primary Data Structure

We assume that N objects are constrained to move on a graph G with E edges and V vertices. The moving objects are updated when they enter or leave edges. They can begin and stop in the middle of edges. A moving object can be represented multiple times on each edge as it leaves and enters edges on its travels. There are a total of n moving object instances (i.e., time intervals on edges) of N moving objects on the entire graph over the time domain $[0, T]$.

The *minGStree* is a combination of strip trees (static part) and interval trees or priority R-tress (dynamic part). The strip trees are used for spatial indexing of the graph edges, and are assumed to fit in main memory. This is a reasonable assumption based on actual road network statistics. For example, the number of edges in the entire road

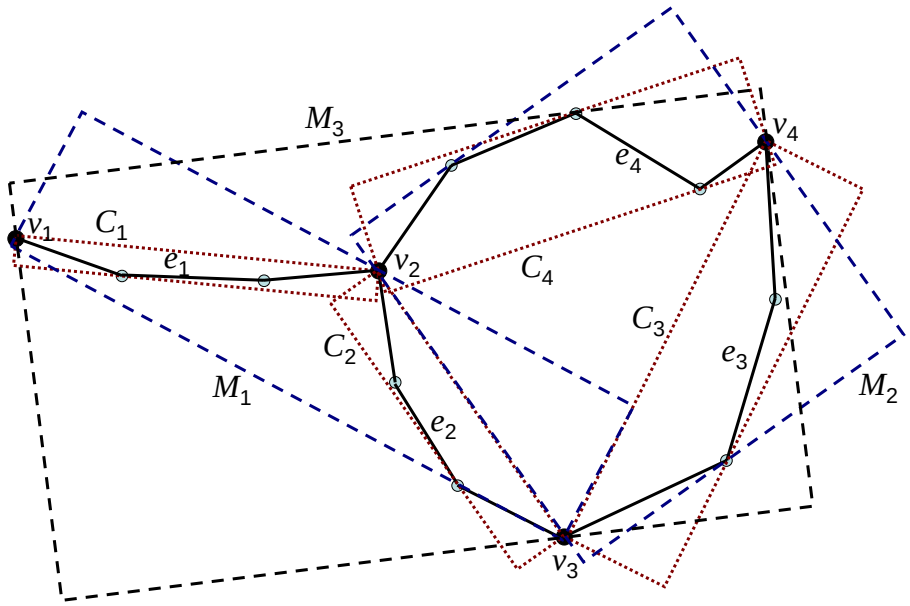


Figure 1: An example graph G with 4 edges $e_1, ..., e_4$ and 4 vertices $v_1, ..., v_4$. The edges are represented as strip trees, with $C_1, ..., C_4$ representing the root bounding boxes for each strip tree. The strip trees are merged bottom up in pairs to construct a graph of strip tress in the *minGStree*.

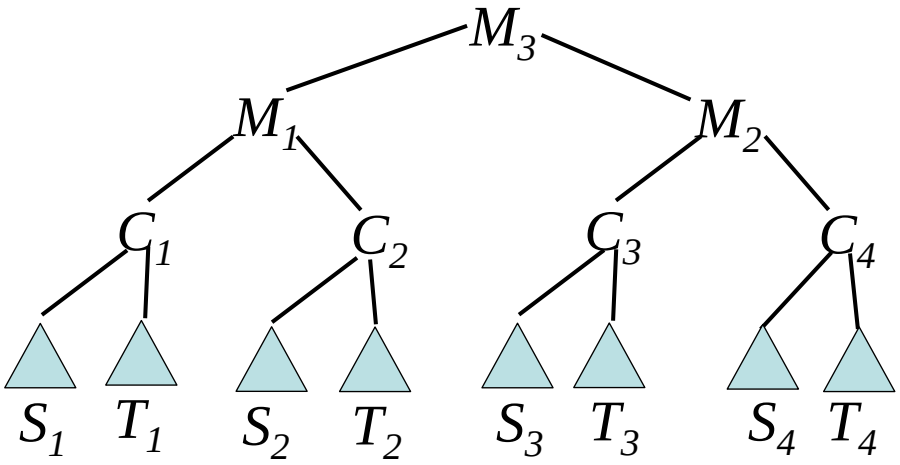


Figure 2: The *minGStree* corresponding to the graph in Figure 1. Each leaf C_i points to strip tree S_i representing edge e_i, and interval tree (or priority R-tree) T_i indexing moving objects on e_i.

network of Canada [14] has E = 1,869,898, with an average of 7.32 segments per edge. If we assume a merged strip tree with E = 2,000,000, with each strip tree requiring 1,000 bytes (a generous allocation), a main memory size of 2 GB will suffice to hold the merged strip tree. In contrast, the number of moving objects through busy intersections is reported [15] as over 500,000 per day, which would result in a much larger space requirement for

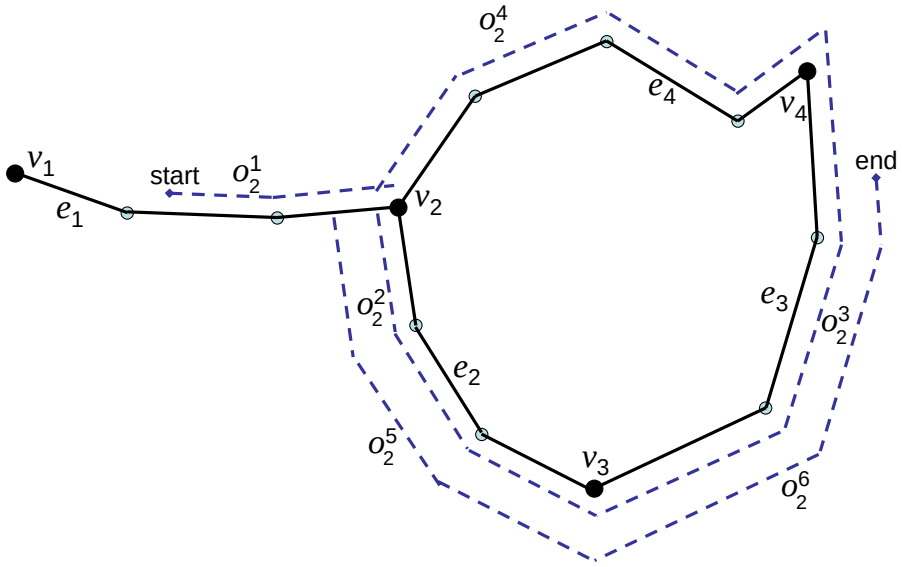


Figure 3: Six moving object instances $o_2^1, ..., o_2^6$ of o_2 moving on the graph from Figure 1.

```
typedef struct {
   no_keys;//number of keys stored in the node
   keys[Bf-1];//values of keys to partition
   ch_pt[Bf];//byte offset of starting positions of the children in the tree file

   left_pt[Bf];// byte offset w.r.t. beginning of left slab list file
   left_no[Bf];//number of items (intervals) stored in each of the Bf left lists
   left_min[Bf];//the minimum value of all left endpoints in each left list

  right_pt[Bf]; // byte offset w.r.t. beginning of right slab list file
  right_no[Bf]; //number of items (intervals) stored in each of the Bf right lists
  right_max[Bf];//the maximum value of all right endpoints in each right list

   multi_pt[(Bf-1)*(Bf-2)/2]; // byte offset w.r.t. beginning of multi slab list file
   multi_no[(Bf-1)*(Bf-2)/2]; //number of items stored in each of the Bf multi lists
}IntervalNode;
```

```
typedef struct {
  no_intvl; ;//number of keys stored in the node
  Interval[B_data];
} LeafNode;
```

```
typedef struct {
  objectid;
  t1, t2;//time interval
  p1,p2;//position interval
} Interval;
```

Figure 4: Interval tree node structures are described using the C language. Bf is the branching factor of the tree. B_data is the number of intervals fitting in a disk block. Intervals in each left slab list are sorted by the increasing order of left end points (i.e., t_1). Intervals in each right slab list are sorted by the decreasing order of right end points (i.e., t_2). Each left, right, and multi slab list stores a set of up to B_data Interval structures.

the interval trees.

Each strip tree S_i (Figure 2) represents a polyline (e.g., a road). The interval trees are used to index the trajectories of objects moving on the graph, and are stored in external memory as external interval trees [3]. Figure 1 illustrates a graph with $V = E = 4$. Figure 2 illustrates the corresponding *minGStree* arising from the graph in Figure 1.

The static part (graph of strip trees) of the *minGStree* is based on the strip tree [4], but it is generalized to allow for indexing collections of strip trees representing a graph. This static part 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. Besides pointing to a strip tree indexing an edge, each leaf node C_i of this binary tree also points to a time interval tree T_i indexing time intervals and storing position intervals of the moving objects on edge e_i, or to a priority R-tree T_i indexing moving object instances as rectangles of time and position intervals.

The rest of this section and Section 5 discuss the dynamic part of the $minGStree_I$ and searching in a $minGStree_I$, respectively. Section 6 discusses the $minGStree_R$ variation.

A moving object instance o_ℓ^j on each edge e_i is described by a time interval $[t_1^j, t_2^j]$ and a position interval $[r_1^j, r_2^j]$. The subscript $\ell \in [1, .., N]$ refers to a unique object identifier, and the superscript $j \in [1, .., n_\ell]$ refers to a unique instance of the moving object o_ℓ at some continuous time interval $[t_1^j, t_2^j]$. We have $n{=}\sum_{\ell=1}^{N} n_\ell$, where n_ℓ is the total number of instances of object o_ℓ moving on edges. Figure 3 illustrates six instances of o_2 moving on the graph from Figure 1. Moving object instances o_ℓ^j are defined as objects appearing on and disappearing from edge e_i, with corresponding updates to T_i. Position intervals $[r_1^j, r_2^j]$ are often [0, 1] (i.e., moving over the entire edge) during a time interval of $[t_1^j, t_2^j]$. We use interval tree T_i to index time intervals of moving objects on edge e_i. Figure 4 displays the node structure of an interval tree. Each time interval $[t_1^j, t_2^j]$ of a moving object instance o_j in an interval tree is store with its corresponding position interval $[r_1^j, r_2^j]$; however, only the time interval is used to create the tree. The position interval is used later in the query step to obtain the velocity of the moving object instance and to check if the current object is in range. Further explanation of the I/O efficient interval tree structure is presented in [3].

Theorem 1. *For a graph containing E edges, and containing n moving object instances over the time domain $[0, T]$ where $n > BE$, the space required for the $minGStree_I$ is $\Theta(E)$ memory cells and $\Theta(\frac{n}{B})$ 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. There is a pointer from nodes C_i to T_i. For c the number of bytes per pointer, the space for these pointers is cE. Therefore, the total space required for the *minGStree* is $O(E)$ memory cells.

Since all n moving object instances are distributed over E edges, they are indexed by E external time interval trees $T_1, .., T_E$ in the $minGStree_I$. An external interval tree indexing g intervals requires $O(\frac{g}{B})$ disk blocks (from Theorem 4.1 of [3]). In interval trees of the $minGStree_I$, each node stores a set of time intervals and associated position intervals, or two intervals for each indexed time interval. Each interval tree T_i thus requires $O(\frac{g}{B})$ disk blocks, for g the number of moving object instances stored for edge e_i. Assuming $n > BE$ and all $g_i > B$, the number of disk blocks used by the E time interval trees is $O(\frac{g_1}{B}) + O(\frac{g_2}{B}) + .. + O(\frac{g_E}{B}) = O(\frac{g_1+g_2+..+g_E}{B}) = O(\frac{n}{B})$, where g_i, $i = 1, .., E$ is the number of moving object instances stored in T_i.

□

Figure 5 shows an example of moving objects for edge e_1 in Figure 1. Figures 6 and 7 illustrates how to index these moving objects on the time interval tree T_1, and how to perform a query on T_i, respectively. Note each moving object has at most one time interval $[t_1, t_2]$ and one position interval $[r_1, r_2]$ stored for each unique time interval completely (or partially) spanning the edge. This assumes that each object is moving at a constant velocity over the time interval $[t_1, t_2]$, and is the minimum information required to determine when a moving object moves on an edge. As the $minGStree_I$ requires $\Theta(E)$ memory cells and $\Theta(n/B)$ disk blocks, it is space optimal assuming constant velocity of objects moving on an edge.

5 Searching in a minGStree$_I$

To answer a time interval query $(R, [t_1, t_2])$, the static part of the $minGStree_I$ is used to find edges intersecting R. If an edge e_i does intersect R, a list F_i of intersecting intervals between e_i and R is constructed. This F_i list will be an input in the next query step. Time interval trees T_i are then used to find objects intersecting in time. With the query time interval $[t_1, t_2]$, searching on a time interval tree produces a list L of moving objects having time intervals intersecting $[t_1, t_2]$. However, before returning L, each element in L is checked to ensure its intersected time interval $[t_{1'}, t_{2'}]$ corresponding position interval $[r_{1'}, r_{2'}]$ actually intersect the query (see Figure 7(d) and Algorithm 6.

Algorithm 1 **InteSearch(**$T_i, F_i, t_1, t_2, queryType$**)** details the search of an I/O efficient interval tree T_i. With the inputs a root node of T_i, a query time interval $[t_1, t_2]$, a list F_i of query position intervals on edge e_i, and a query type *queryType*, whose value is 1, 2, or 3 if $[t_1, t_2]$, only t_1, or only $t_2 \in$ the current node boundary, respectively, the output of this algorithm is a list L of moving objects in range of tree T_i. Line 9 of Algorithm 1 calls **SearchIn(**$T_i, F_i, t_1, t_2, queryType$**)**, which reports objects in range of the current node of T_i. Lines 10 to 32 of Algorithm 1 recursively call the function itself for children

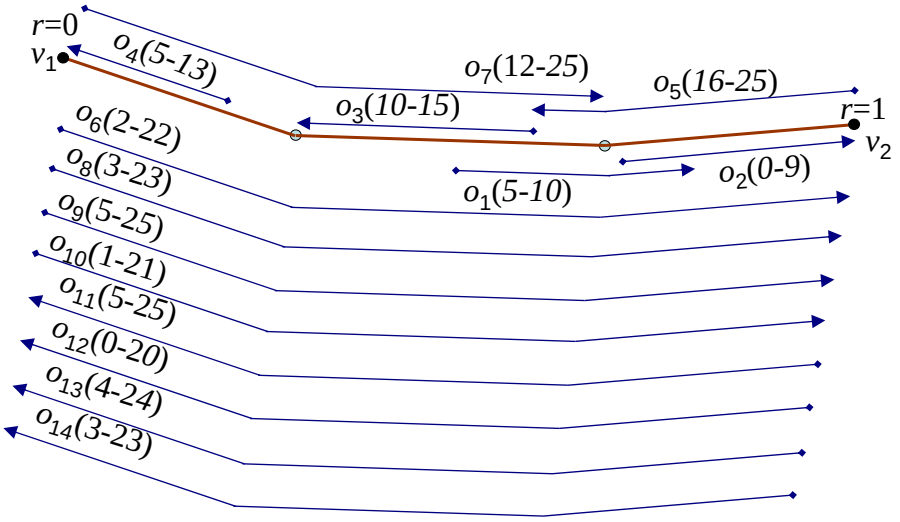
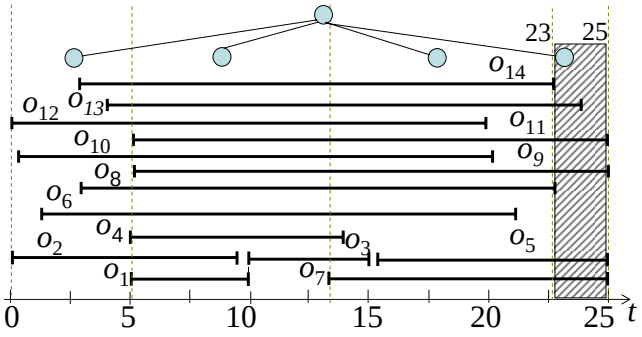


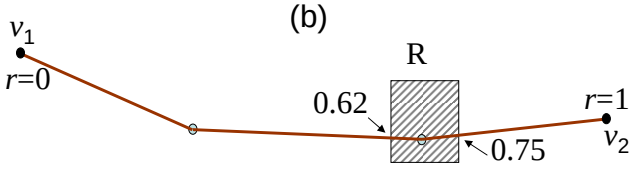
Figure 5: An example of 14 moving objects on edge e_1 of the graph in Figure 1. There are 6 moving objects having their begin or end points falling inside the edge. Eight moving objects $o_6, o_8, ..., o_{14}$ move across the entire edge. Each directed polyline represents a position interval of a moving object with a direction. The two numbers in parentheses represent time intervals of corresponding moving objects. We assume that moving objects on the same edge have the same velocity.

id	$[t_1,t_2]$	$[r_1,r_2]$
o_1	5 10	0.5 0.8
o_2	0 9	0.7 1
o_3	10 15	0.6 0.4
o_4	5 13	0.3 0
o_5	16 25	1 0.6
o_6	2 22	0 1
o_7	12 25	0 0.68
o_8	3 23	0 1
o_9	5 25	0 1
o_{10}	1 21	0 1
o_{11}	5 25	1 0
o_{12}	0 20	1 0
o_{13}	4 24	1 0
o_{14}	3 23	1 0

(a)



(b)



(c)

Figure 6: An example time interval tree T_1 (b) for 14 moving objects on e_1 described by table (a), another representing of moving objects from Figure 5. In the table (a), each object is depicted by a time interval and a position interval. The dashed vertical lines in (b) indicate slab boundaries for an external interval tree. (c) shows an example of a Q_2 query, where rectangle R intersects edge e_1 in Figure 1 at the position interval $[r_1, r_2]$=[0.62, 0.75] and the time interval query is $[t_1, t_2]$=[23, 25].

Algorithm 1: **InteSearch**($T_i, F_i, t_1, t_2, queryType$)

The general algorithm for interval tree searching on one edge e_i.

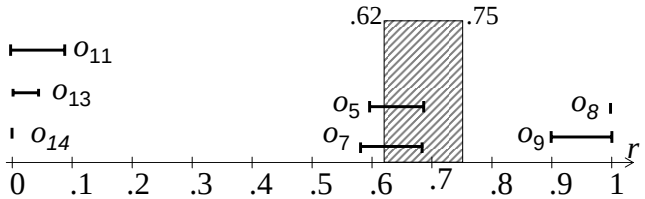
input : Adrress of the root node of T_i, query time interval $[t_1, t_2]$, and F_i: list of query position intervals on edge e_i

output: a list L of moving objects in range

```
1.1  begin
1.2      //queryType ← 1,2, or 3 if [t1,t2], only t1, or only t2 ∈ current node boundary
1.3      Read the current node Ti from disk
1.4      L ← ∅
1.5      if Ti is a leaf then
1.6          foreach [t1^j, t2^j]_k ∈ Ti.intervalList, k ∈ [1, |Ti.intervalList|] do
1.7              L ← L ∪ IntPosCheck([t1^j, t2^j]_k, [r1^j, r2^j]_k, t1, t2, Fi)
1.8      else
1.9          L ← L ∪ SearchIn(Ti, Fi, t1, t2, queryType)//report items at current node
1.10         kl ← key boundary of t1
1.11         kr ← key boundary of t2
1.12         if queryType = 1 then
1.13             if kl = kr then
1.14                 //e.g., δ node in Figure 8
1.15                 L ← L ∪ InteSearch(Ti.child[kl], Fi, t1, t2, queryType)
1.16             else
1.17                 //e.g., β node in Figure 8
1.18                 L ← L ∪ InteSearch(Ti.child[kl], Fi, t1, t2, 2)
1.19                 L ← L ∪ InteSearch(Ti.child[kr], Fi, t1, t2, 3)
1.20                 for w = kl + 1 to kr − 1 do
1.21                     L ← L ∪ retrieveItems(Ti.child[w])
                         //report items in the middle part;
1.22         else
1.23             if queryType = 2 then
1.24                 //e.g., γL node in Figure 8
1.25                 L ← L ∪ InteSearch(Ti.child[kl], Fi, t1, t2, 2)
1.26                 for w = kl + 1 to Ti.lastBoundary do
1.27                     L ← L ∪ retrieveItems(Ti.child[w])
                         //report items in the middle part;
1.28             else
1.29                 //queryType = 3 e.g., γR node in Figure 8
1.30                 L ← InteSearch(Ti.child[kr], Fi, t1, t2, 2)
1.31                 for w = 0 to kr − 1 do
1.32                     L ← L ∪ retrieveItems(Ti.child[w])
                         //report items in the middle part;
1.33     return L;
1.34 end
```

id	$[t_{1'}, t_{2'}]$	$[r_{1'}, r_{2'}]$
o_{14*}	23 23	0 0
o_{13*}	23 24	0.05 0
o_{11*}	23 25	0.1 0
o_9	23 25	0.9 1
o_8	23 23	1 1
o_7	23 25	0.58 0.68
o_{5*}	23 25	0.69 0.6

(d)



(e)

Figure 7: Following Figure 6, (d) lists objects falling in range during time interval [23, 25] with their intersected time intervals and the corresponding position intervals at those intersected time intervals. (e) shows that with those intervals obtained in (d), two moving object instances p_5 and p_7 are in range with the query at Figure 6(c). Note $*$ in the first column of the table (d) means that objects move from v_2 to v_1 instead of v_1 to v_2, giving $r_2 > r_1$.

of T_i based on *queryType*. Algorithms 3 and 4 describe two functions **checkRightList** and **reportRightList**, respectively, called by Algorithm 2. Since intervals in a right slab list are sorted by the decreasing order of t_2^j of all moving object o_j in this right slab list, **checkRightList** selects intervals having $t_2^j \geq t_2$ to ensure $[t_1^j, t_2^j]$ intersecting the query time $[t_1, t_2]$. **IntersectPosInt** is then called for each intersected $[t_1^j, t_2^j]$ to check the intersection of the query position. With **reportRightList**, all intervals in the input right slab list already intersected to the query time $[t_1, t_2]$, the algorithm goes through all of those intervals to report objects having position intervals intersecting to F_i.

Note that **IntersectPosInt**$([t_1^j, t_2^j], [r_1^j, r_2^j], t_1, t_2)$ called by Algorithm 6 returns an intersected position interval $(r_{1'}^j, r_{2'}^j)$ from a time interval, a position interval, and a query time interval (t_1, t_2) using equation 2. Based on the time interval $[t_1^j, t_2^j]$ and position interval $[r_1^j, r_2^j]$, the velocity vel_j of a moving object instance o_j on edge e_i can be computed as follows:

$$vel_j = \frac{(r_2^j - r_1^j)|e_i|}{t_2^j - t_1^j} \tag{1}$$

where $|e_i|$ is the length of edge e_i. When $r_2^j \geq r_1^j$, o_j moves toward the direction from $r = 0$ to $r = 1$, and vice versa. The value vel_j can be equal to, greater than, or less than 0. From here, we can compute an arbitrary position r of o_j at a time $t \in [t_1^j, t_2^j]$ on edge e_i by $r = r_1^j + (vel_j * \triangle t)/|e_i|$, where $\triangle t$=$(t - t_1^j)$. This formula of r can be simplified to the convex combination

$$r = \frac{r_1^j(t_2^j - t) + r_2^j(t - t_1^j)}{(t_2^j - t_1^j)} \tag{2}$$

A moving object in L is in range if the corresponding position interval $[r_{1'}, r_{2'}]$ of its intersected time $[t_{1'}, t_{2'}]$ intersects with at least one of the intervals in F_i. For example, moving object o_7 (Figure 6) has $[t_1^7, t_2^7]$=[12, 25], $[r_1^7, r_2^7]$=[0, 0.68], giving r=0.58 for t=23,

Algorithm 2: **SearchIn**($T_i, F, t_1, t_2, queryType$)

Algorithm for searching objects at current node T intersecting time query t_1, t_2 and query positions in F

input : Adrress of the root node of T_i, query time interval $[t_1, t_2]$, list F of query position intervals on the current edge, and *queryType* carrying the type of $[t_1, t_2]$ on the current node

output: a list L of moving objects in range

```
2.1  begin
2.2      L ← ∅
2.3      k_l ← key boundary of t_1
2.4      k_r ← key boundary of t_2
2.5      if queryType = 1 then
2.6          L ← L ∪ checkRightList(T_i, k_l, t_1, t_2, F)
2.7          L ← L ∪ checkLeftList(T_i, k_r, t_1, t_2, F)
2.8          L ← L ∪ reportMultiList(T_i, k_l, k_r, t_1, t_2, F)
2.9          if k_l ≠ k_r then
2.10             for i = k_l to k_r − 1 do
2.11                 L ← L ∪ reportLeftList(T_i, i, t_1, t_2, F)
2.12             L ← L ∪ reportRightList(T_i, k_r, t_1, t_2, F)
2.13     else
2.14         if queryType = 2 then
2.15             L ← L ∪ checkRightList(T_i, k_l, t_1, t_2, F)
2.16             L ← L ∪ reportMultiList(T_i, k_l, k_r, t_1, t_2, F)
2.17             for i = k_l to k_r − 1 do
2.18                 L ← L ∪ reportLeftList(T_i, i, t_1, t_2, F)
2.19         else
2.20             L ← L ∪ checkLeftList(T_i, k_r, t_1, t_2, F)
2.21             L ← L ∪ reportMultiList(T_i, k_l, k_r, t_1, t_2, F)
2.22             for i = k_l to k_r do
2.23                 L ← L ∪ reportRightList(T_i, i, t_1, t_2, F)
2.24     return L;
2.25 end
```

and r=0.68 for t=25 using equation 2 for r. Thus, $[r_{1'}^7, r_{2'}^7]$=[0.58, 0.68].

checkLeftList and **reportLefttList** in Algorithm 2 are similar to **checkRightList** and **reportRightList**, respectively. Note that intervals in a left slab list are sorted in the increasing order of t_1^j; thus the condition in the while loop of **checkLeftList** is $t_1^j \leq t_1$ to ensure $[t_1^j, t_2^j]$ intersecting the query time $[t_1, t_2]$. **reportMultiList** in Algorithm 5 is

Algorithm 3: **checkRightList**$(T, slab, t_1, t_2, F)$

The algorithm for finding objects falling in range in a right slab list of the current tree node T.

input : Tree node T, slab key *slab* of T, query time interval $[t_1, t_2]$, and F: list of query position intervals on edge e_i

output: List of objects falling in range

```
3.1 begin
3.2   Read right slab list S_L from disk at address T.rightSlab[slab]
3.3   L ← ∅
3.4   while t_2^j ≥ t_1, where[t_1^j, t_2^j]_k ∈ S_L do
3.5     L ← L ∪ IntPosCheck([t_1^j, t_2^j]_k, [r_1^j, r_2^j]_k, t_1, t_2, F)
3.6   return L;
3.7 end
```

Algorithm 4: **reportRightList**$(T, slab, t_1, t_2, F)$

The algorithm for finding objects falling in range in a right slab list of the current tree node T.

input : Tree node T, slab key *slab* of T, query time interval $[t_1, t_2]$, and F: list of query position intervals on edge e_i. Note that objects in the right slab list at *slab* already intersect $[t_1, t_2]$.

output: List of objects falling in range

```
4.1 begin
4.2   Read right slab list S_L from disk at address T.rightSlab[slab]
4.3   L ← ∅
4.4   foreach [t_1^j, t_2^j]_k ∈ S_L, k ∈ [1, |S_L|] do
4.5     L ← L ∪ IntPosCheck([t_1^j, t_2^j]_k, [r_1^j, r_2^j]_k, t_1, t_2, F)
4.6   return L;
4.7 end
```

used to reporting objects falling in range in multi slab lists.

In the theorem below, we assume that the $minGStree_I$ "upper part" (i.e. all except the external interval trees) can be stored in main memory. Thus, no I/Os are required to search precisely which strip trees S_i are intersected, and where.

Objects are moving continuously. The number of objects entering an edge at one time (instant) or leaving an edge at one time is constrained by edge capacity. We assume that only one moving object can enter and leave an edge at a time. In a time interval tree, moving object instances o_ℓ^j are unique.

Lemma 1. *The expected number of intervals in a time interval tree T_i is $O(\frac{n}{E})$.*

Algorithm 5: **reportMultiList**$(T, slab_L, slab_R, t_1, t_2, F)$

The algorithm for reporting objects falling in range in multi slab lists intersecting slab [$slab_l$, $slab_R$] of the current tree node T.

input : Tree node T, two slab keys $slab_L$ and $slab_R$ of T, query time interval $[t_1, t_2]$, and F: list of query position intervals

output: List of objects falling in range

```
5.1 begin
5.2     L ← ∅
5.3     foreach multislab list M intersects slab[slab_L, slab_R] of current node T do
5.4         Read M from disk
5.5         foreach [t_1^j, t_2^j]_k ∈ M, k ∈ [1, |M|] do
5.6             L ← L ∪ IntPosCheck([t_1^j, t_2^j]_k, [r_1^j, r_2^j]_k, t_1, t_2, F)
5.7     return L;
5.8 end
```

Algorithm 6: **IntPosCheck**$([t_1^j, t_2^j], [r_1^j, r_2^j], t_1, t_2, F)$

The algorithm for checking if a current moving object is in range.

input : Time interval $[t_1^j, t_2^j]$ and position interval $[r_1^j, r_2^j]$ of object o_j, query time interval $[t_1, t_2]$, and F: list of query intervals on edge e_i

output: return o_j if intersected position interval of o_j is in range; otherwise; return empty

```
6.1 begin
6.2     (r_1'^j, r_2'^j) ← IntersectPosInt([t_1^j, t_2^j], [r_1^j, r_2^j], t_1, t_2)
6.3     if IntersectCheck(r_1'^j, r_2'^j, F) then
6.4         return o_j;
6.5     return ∅;
6.6 end
```

Proof. We assume objects are moving at a constant velocity $vel \in [vel_{min}, vel_{max}]$ per edge. Initially, N moving objects are distributed in a uniform random fashion over the E edges at a densityof η per unit distance. As moving objects arrive at a vertex v_i, they are redirected uniform randomly into one of the $|v_i|$ edges connected to v_i. As time $t \rightarrow T$, and within a constant factor $\rightarrow \frac{vel_{max}}{vel_{max}}$, the number of moving object instances flowing through short edges is equal to the number of moving object instances flowing through long edges over the time interval $[0, T]$. For n moving object instances, as time $t \rightarrow T$, where $T > max(|e_i|)/vel_{min}$, the number of moving object instances per edge $\rightarrow \frac{n}{E}$. Thus, each time interval tree T_i stores $O(\frac{n}{E})$ time intervals, one for each moving object instance. □

The above model is reasonable for the purpose of simulating objects moving on a graph (e.g., vehicles on a road network). If moving objects stop moving for a long period, our assumption about the number of objects leaving an edge using the same for all edges does not hold.

Lemma 2. *On an interval tree T_i containing g intervals, performing a $Q_2 = (R, [t_1, t_2])$ query takes $O(\log_B g + k)$ I/Os, where k is the number of disk blocks required to store moving object instances intersecting $[t_1, t_2]$.*

Proof. From Theorem 4.1 of [3], a stabbing query on an interval tree containing g intervals requires $O(\log_B g + k')$ I/Os, where k' is the number of disk blocks required to store the answer (i.e., time intervals intersecting $[t_1, t_2]$). At each of k' intersected time intervals, its corresponding intersected position interval is checked for the intersection with R. This step takes zero I/Os. Note that the number of moving objects in range is k'', which is smaller than or equal to k'. A range query requires I/Os for two stabbing queries at t_1 and t_2, plus I/Os for intersecting intervals stored in tree nodes in the *'middle part'* (shaded area in Figure 8). We call the two leaf nodes that stabbing queries at t_1 and t_2 reach β_L and β_R, respectively. Assume β is the nearest common ancestor of β_L and β_R. The *'middle part'* contains all nodes of the interval tree that fall between the two paths from β to β_L and β to β_R. All intervals stored in the *'middle part'* nodes intersect $[t_1, t_2]$, which require reading all the *'middle part'* nodes from disk. If the stabbing query at t_1 requires $O(\log_B g + k_1)$ I/Os, the stabbing query at t_2 requires $O(\log_B g + k_2)$ I/Os, and reading the *'middle part'* nodes requires k_3 I/Os, the number of I/Os required for a range query is $O(\log_B g + k_1 + \log_B g + k_2 + k_3) = O(\log_B g + k)$, where k is the number of disk blocks required to obtain time intervals intersecting the query $[t_1, t_2]$. □

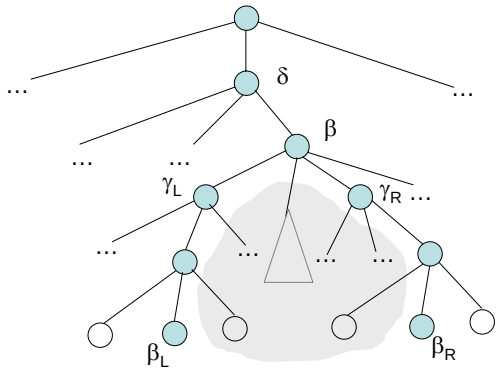


Figure 8: Illustration of external interval tree nodes in a range query $[t_1, t_2]$. β_L and β_R are the two leaf nodes that two stabbing queries at t_1 and t_2 reach. β is the nearest common ancestor of β_L and β_R. The shaded area contains nodes in the *'middle part'* between two stabbing query paths β to β_L and β to β_R.

Theorem 2. *For n moving object instances randomly distributed on the E edges of a graph, the number of I/Os required to determine the moving objects intersecting one edge at time t_q (or at time interval $[t_{q1}, t_{q2}]$) in a minGStree$_I$ is expected to be $O(\log_B \frac{n}{E} + k)$, where k is the number of disk blocks required to store the time intervals intersecting the query time $[t_1, t_2]$.*

Proof. The expected number of intervals in T_i is $O(\frac{n}{E})$ (Lemma 1), and searching on T_i takes $O(\log_B \frac{n}{E} + k)$ I/Os (Lemma 2). □

Note that the number of moving object instances in range k'' is smaller than or equal to k (the number of intersecting time intervals). In the worst case $k'' \to 0$ when none of the moving objects on e_i intersects query rectangle R during query time interval $[t_1, t_2]$.

Theorem 3. *For n moving object instances randomly distributed on the E edges of a graph, the number of I/Os required to determine the moving objects intersecting a Q_2 query in a minGStree$_I$ is expected to be $O(\log_B \frac{n}{E} + k)$, where k is the number of disk blocks required to store the time intervals intersecting the query time $[t_1, t_2]$.*

Proof. Strip trees S_i, stored in the main memory, are used to determine if R intersects an edge e_i. This step takes zero I/Os. From Theorem 2 searching on one edge requires $O(\log_B \frac{n}{E} + k)$ I/Os. If R of Q_2 intersects D edges of the graph, searching on D edges takes $O(D \log_B \frac{n}{E} + k)$. Assuming D is a constant, much smaller than E, the expected number of I/Os required to answer a Q_2 query is $O(\log_B \frac{n}{E} + k)$, where k is the number of disk blocks required to store the time intervals intersecting the time query. □

In the worst case, when R intersects all E edges, the number of I/Os required to determine moving objects falling in range is $O(E \log_B \frac{n}{E} + k)$. However, in the next corollary, we will show that this bound is dominated by k.

Corollary 4. *Assume at most E^c moving objects N on the graph, for c a small constant and greater than 1. When rectangle R intersects all E edges and the time domain of the minGStree$_I$ $[0, T] \subset [t_1, t_2]$, the number of I/Os required to answer this query is $O(E \log_B \frac{n}{E} + k)$. This expression is dominated by k, where k is the number of disk blocks required to store the time intervals intersecting the time query.*

Proof. Assume each moving object travels on $O(E)$ edges. Thus, the number of moving object instances $n = O(E^{c+1})$. If all moving object instances of one edge e_i fall in range, we expect $k_i = \frac{n}{EB}$. For R intersecting E edges, the number of I/Os expected is $E \times O(\log_B \frac{n}{E} + k_i) = O(E \log_B \frac{n}{E} + E \times \frac{n}{EB}) = O(E \log_B \frac{n}{E} + \frac{n}{B}) = O(E \log_B \frac{E^{c+1}}{E} + \frac{E^{c+1}}{B})$ $= O(E \log_B E^c + \frac{E^{c+1}}{B}) = O(E(\log_B E + \frac{E^c}{B}))$. For large graphs, we assume B is always smaller than E, so $\log_B E$ is always smaller than $\frac{E^c}{B}$, or $E \log_B E$ is always smaller than $\frac{E^{c+1}}{B}$. Therefore, the number of I/Os k dominates the expression $O(E \log_B \frac{n}{E} + k)$.

For reasonable assumptions about the size of N and E, the number of I/Os required in the worst case is dominated by k, the number of blocks of external memory required to store the time intervals intersecting the time query.

□

Corollary 5. *Inserting or deleting a moving object instance into or from the $minGStree_I$ is expected to require $O(log_B \frac{n}{E})$ I/Os amortized.*

Proof. Insertion or deletion of a moving object instance in the $minGStree_I$ happens in one time interval tree T_i From [3], we know that the amortized I/Os required for updating an I/O interval tree are $O(log_B \frac{n}{E})$. □

For $R \cap G = \emptyset$, or $[t_1, t_2] \cap [0, T] = \emptyset$, we can trivially determine that the query answer is $\emptyset$ with 0 I/Os.

6 A minGStree Variation

As an interesting variation, we define the $minGStree_R$ where all time interval trees in the $minGStree$ can be replaced by priority R-trees [2]. Each moving object instance o_ℓ^j described by a 4-tuple $[t_1^j, t_2^j], [r_1^j, r_2^j]$ is treated as a rectangle R_ℓ^j. As the construction and the searching algorithm on an priority R-tree are detailed in [2], we do not repeat those here. We focus on the combination of priority R-trees with a graph of strip trees to form the $minGStree_R$.

Assuming B is the number of rectangles that can fit in one disk block, rectangles are grouped in sets of size B to form the leaves of a priority R-tree T_i. From Lemma 1, we expect the number of rectangles stored in one priority R-tree T_i to be $O(\frac{n}{E})$. For efficiency, a packing heuristic is used to maximize the overlap among the rectangles stored in one leaf.

An R-tree indexing g rectangles occupies $\Theta(g/B)$ disk blocks (Section 1.1 of [2]). Therefore, Theorem 1 holds for the $minGStree_R$ variation. In other words, for a graph containing E edges, and containing n moving object instances over the time domain $[0, T]$, the space required for the $minGStree_R$ is $\Theta(E)$ memory cells and $\Theta(\frac{n}{B})$ disk blocks, for B the number of elements transmitted by one external memory access.

From Theorem 2 of [2] and Lemma 1, we arrive at the following claim.

Theorem 6. *For n moving object instances randomly distributed on the E edges of a graph, the number of I/Os required to determine the moving objects intersecting one edge at time t_q (or at time interval $[t_{q1}, t_{q2}]$) in a $minGStree_R$ is expected to be $O(\sqrt{\frac{n}{EB}} + k''')$, where k''' is the number of disk blocks required to store the rectangles intersecting the query.*

Using the same arguments as in the proof of Theorem 3 (i.e., a small number of edges e_i intersect the query rectangle R), we have the following claim.

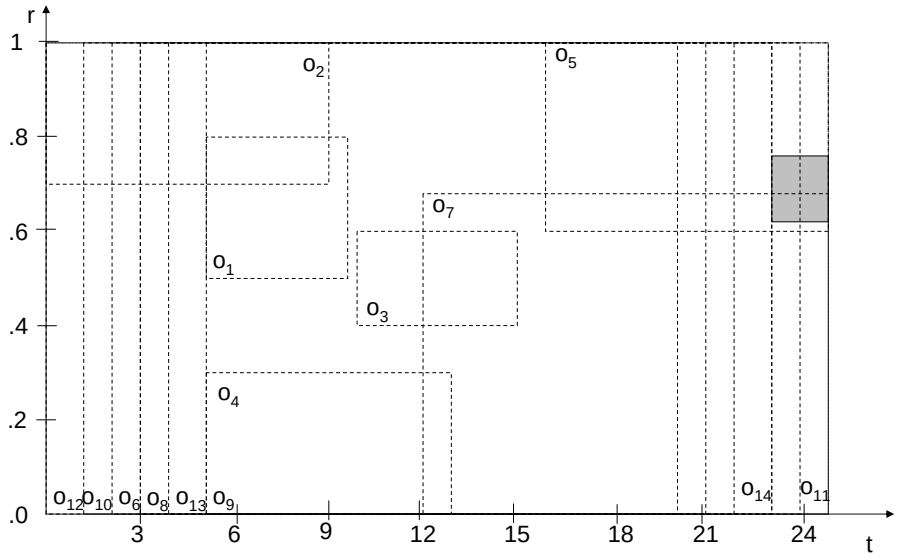


Figure 9: Rectangles in dashed lines illustrating the $[t_1^j, t_2^j], [r_1^j, r_2^j]$ rectangles of 14 moving object instances in Figure 6(a). Five rectangles of moving objects o_5, o_7, o_9, o_{11}, and o_{13} intersect with the shaded query Q_3 = ([23,25],[0.62,0.75]) in Figure 6(c). Only two moving objects o_5 and o_7 are actually in range as shown in Figure 7(e).

Theorem 7. *For n moving object instances randomly distributed on the E edges of a graph, the number of I/Os required to determine the moving objects intersecting a Q_2 query in a $minGStree_R$ is expected to be $O(\sqrt{\frac{n}{EB}} + k''')$, where k''' is the number of disk blocks required to store the rectangles intersecting Q_2.*

In the worst case, when R intersects all E edges, the number of I/Os required to determine moving objects falling in range is $O(E\sqrt{\frac{n}{EB}} + k''')$, or $O(\sqrt{\frac{nE}{B}} + k''')$.

Note that k''' is likely to be larger than k'', the number of disk blocks required to store the query answer, but less than k from the searching complexity $O(log_B \frac{n}{E} + k)$ of the $minGStree_I$. For example, the query in Figure 6(c) has two moving objects in range as shown in Figure 7(e). The $minGStree_I$ search has seven time intervals in range (see Figure 7(d)). The $minGStree_R$ search reports five rectangles in range (see Figure 9). There is a trade-off between the $O(log_B \frac{n}{E} + k)$ I/Os required by the $minGStree_I$ and the $O(\sqrt{\frac{n}{EB}} + k''')$ I/Os required by the $minGStree_R$.

7 Ordered Polyline Representation

The problem of objects moving at constant velocity on the edges of graphs is more precisely represented by a diagonal line segment of the $[t_1, t_2] \times [r_1, r_2]$ rectangle for each moving object on each edge. Each point on this line segment corresponds to a position of moving object at a specific time. If a line segment intersects a rectangle query, its corresponding

moving object is in range. Our problem is now how to index line segments efficiently to achieve an efficient search on moving objects. Figure 10 shows an example of 18 objects moving at constant velocity over an edge.

Ordered polylines p_i are created by connecting parts of line segments formed from their intersection (with each other and with the $r = 0$, and $r = 1$ boundary). For example, the first two ordered polylines in Figure 10 are $p_1 = \{o_{1.1}, o_{2.2}\}$ and $p_2 = \{o_{2.1}, o_{1.2}\}$. Ordered polylines are arranged as a balanced search tree based on each p_i dividing the space. Points to the left of p_i are guaranteed to be in the left subtree of the node containing p_i; similarly points to the right of p_i are in the right subtree of the node containing p_i.

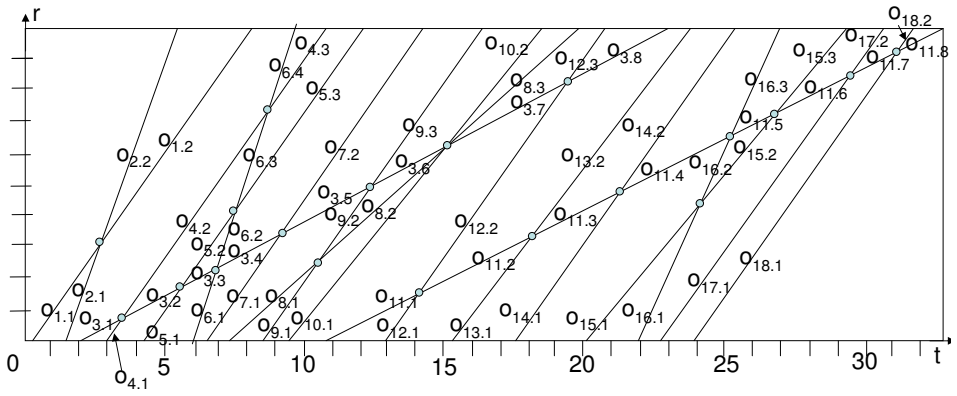


Figure 10: Line segments represent 18 moving objects traveling at a constant velocity over an edge.

In the worst case, every line segment representing a moving object instance intersects the line segments representing all other moving object instances on the same edge (see Figure 11). For g_i moving object instances on edge i, this worst case results in $O(g_i^2)$ line segments, with each ordered polyline requiring $O(g_i)$ line segments. The number of ordered polylines is still precisely g_i. We thus need $O(log(g_i))$ time to find which line segments of a single ordered polyline intersect the Q_2 query.

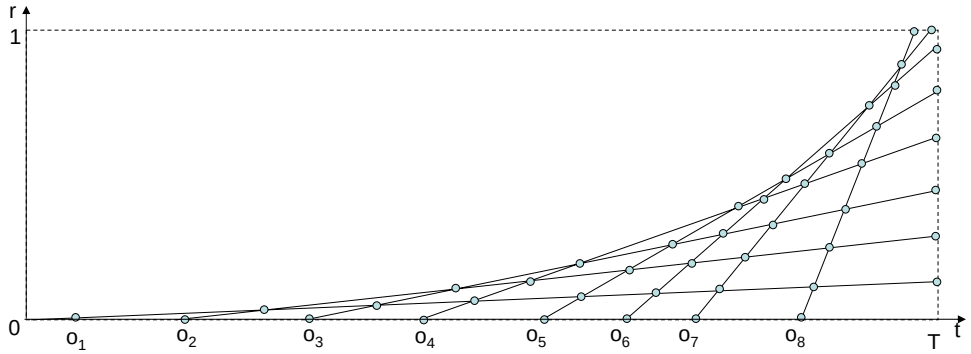


Figure 11: Example of 8 polylines representing 8 moving object instances o_1, ..., o_8 in the worst case.

Theorem 8. *Time to search for line segments of a single ordered polyline intersecting a Q_2 query is $O(log(g_i) + a)$, for a the number of line segments intersecting Q_2.*

Proof. In order to index an ordered polyline (see Figure 12), we use a one-dimensional range tree to index its points. Since points (t, r) in a polyline are ordered increasing on both t and r values, we can choose either t or r as the key of the range tree without changing its structure. In our range tree, a leaf node of the tree contains two values t and r of a point, and an internal node contains two average values ($\bar{t}$,$\bar{r}$), which are the average of the (t, r) values in the rightmost left node in the left subtree and the leftmost leaf node in the right subtree. All leaf nodes are threaded. Figure 13 shows the range tree indexing the ordered polyline in Figure 12.

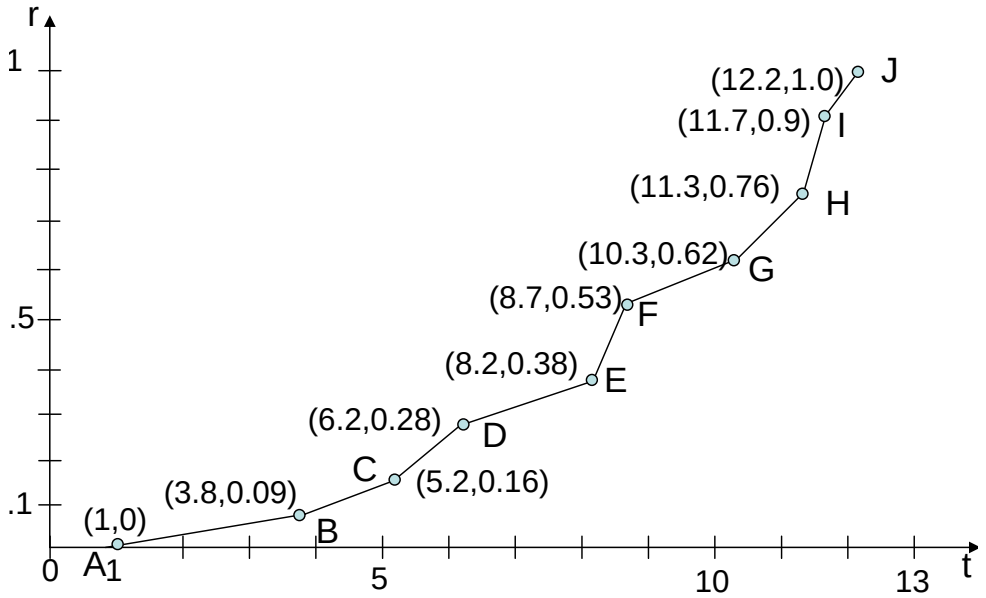


Figure 12: Example of an ordered polyline containing 10 points A, B,.., J.

Searching for segments of an ordered polyline intersecting a query $Q_3 = [t_1, t_2] \times [r_1, r_2]$ starts from the root of the range tree, then go to the left child or the right child based on the (t, r) values. When a leaf node is reach, we follow the threads from the current leaf to check for the intersection between the query and line segments.

At an internal node, the $(\bar{t}, \bar{r})$ values are compared to the $[t_1, t_2]$ and $[r_1, r_2]$ ranges of the query rectangle Q_3. We follow the proper child based on the following four choices:

1. If the $\bar{r}$ ($\bar{t}$) value of the current node falls in the $[r_1, r_2]$ ($[t_1, t_2]$), and the $\bar{t}$ ($\bar{r}$) value does not, we travel down based on the $\bar{t}(\bar{r})$ value.

2. If both $\bar{t}$ and $\bar{r}$ values fall outside $[t_1, t_2]$ and $[r_1, r_2]$, respectively, and result in travel to the same child, we follow that child.

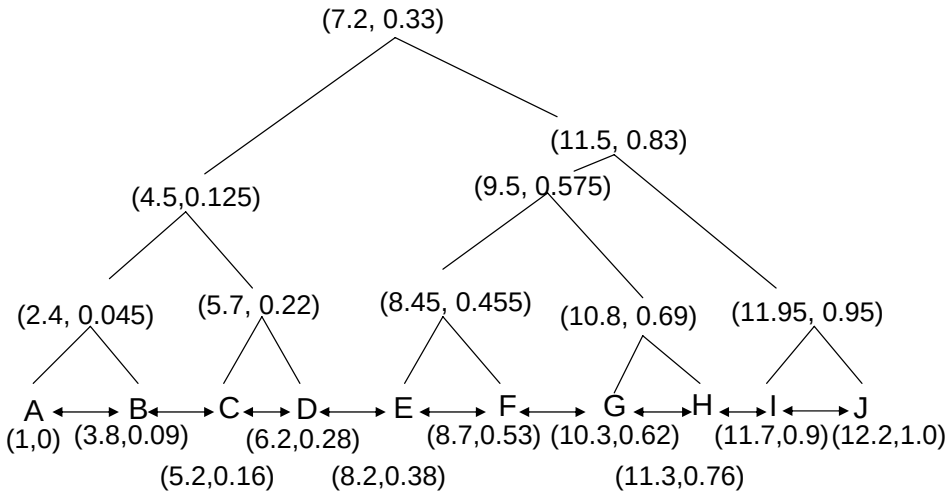


Figure 13: Example of a range tree indexing the ordered polyline shown in Figure 12.

3. If both $\bar{t}$ and $\bar{r}$ values fall outside $[t_1, t_2]$ and $[r_1, r_2]$, respectively, and $\bar{t}$ and $\bar{r}$ values make us travel down to different sides of the current node, we halt.

4. If both $\bar{t}$ and $\bar{r}$ values fall inside $[t_1, t_2]$ and $[r_1, r_2]$, respectively, we travel down one side (e.g., left side) of the current node until reaching a leaf node l. We then check the intersection of the segments threaded from l (e.g., to the right direction) until we reach a line segment that does not intersect Q_3.

From the above algorithm, we see that in the worst case, when the root node meets the fourth choice above, the time required is $log_2(g_i) + a$ or $O(log_2(g_i) + a)$, where g_i is the number of points in the ordered polyline p_i, and a is the number of line segments intersecting Q_3.

□

Theorem 9. *For a single edge e_i containing g_i moving object instances, the time to answer a Q_2 query in the worst case is $O(log(g_i)^2 + k)$, where k is the number of ordered polylines in range.*

Proof. An ordered polyline tree indexes ordered polylines p_i as a balanced search tree based on each p_i diving the space. Each internal node contains a range tree indexing line segments of an ordered polyline used to divide space, and points to two other children indexing dividing ordered polylines. Each leaf node contains a range tree for an ordered polyline. Figure 14 shows an example of the ordered polyline tree indexing the 18 ordered polylines shown in Figure 10.

Since an ordered polyline tree is a balanced search tree, and indexes g_i ordered polylines p_i, its height is $log_2 g_i$. From theorem 8, searching at each range tree at each node on the ordered polyline tree takes $O(log(g_i))$ time in the worst case. Therefore, searching on the ordered polyline tree for ordered polylines intersecting Q_2 requires $O(log_2(g_i) \times log(g_i) + k)$,

or $O(log(g_i)^2 + k)$ time in the worst case, where k is the number of ordered polylines in range. □

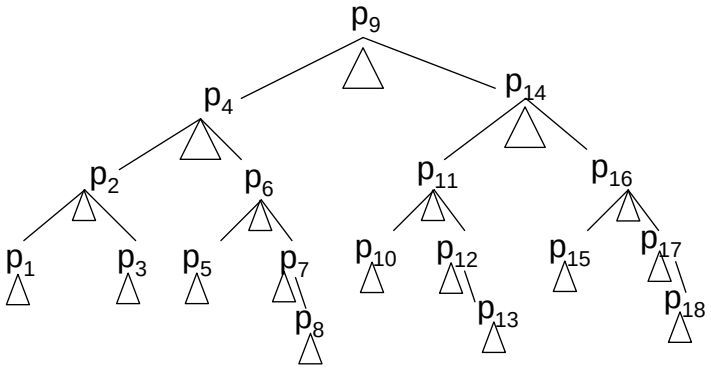


Figure 14: Example of the ordered polyline tree indexing the 18 ordered polylines shown in Figure 10. Ordered polyline $p_9 = \{o_{9.1}, o_{8.2}, o_{8.3}\}$.

Note that in the best case, when each ordered polyline is a line segment, searching on a range tree for this line segment requires $O(1)$ time; thus, searching on the ordered polyline tree only requires $O(log(g_i) + k)$, where k is the number of ordered polylines in range. The space for an ordered polyline in the best case is $O(g_i)$, versus in the worst case (see Figure 11) where the space required is $O((g_i)^2)$.

8 Conclusion

We present a new data structure, the *minGStree*, for efficient search of moving objects (e.g. vehicles) on planar (or non-planar) graphs. The *minGStree* is a combination of strip trees and interval trees. Strip trees are used for spatial indexing of the graph edges. Each strip tree (at leaf level) represents a polyline (corresponding to a road or road segment in a road network). The interval trees (in the $minGStree_I$ version) or priority R-trees (in the $minGStree_R$ version) are used to index the trajectories of moving objects on graphs indexed by strip trees. There are some advantages for the *minGStree*. First, the top strip trees indexing the graph geometry, and the bottom interval trees indexing only time intervals of moving objects are independent. For example, one can update the bottom trees (interval trees or priority R-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 single edge; for example, how many vehicles move on a specific road at a specific time or during a specific time interval.

The *minGStree* is directly implementable as an I/O efficient data structure by replacing the internal memory interval trees with the optimal external memory interval

trees, or priority R-trees. It remains to experimentally validate the *minGStree* with an implementation of I/O-efficient external memory interval trees (e.g. [6]).

It is an open problem whether the expected number of I/Os required to answer a Q_1 or Q_2 query can be made output sensitive (i.e. k is precisely the number of disk blocks required to store the answer). Another open problem is how to efficiently index ordered polylines representing moving object instances so as to achieve an I/O efficient worst case optimal search complexity, or how to build an I/O efficient data structure to achieve a polylogarithmic search time on a set of g_i moving object instances.

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

[2] L. Arge, M. de Berg, H. J. Haverkort, and K. Yi. The priority r-tree: A practically efficient and worst-case optimal r-tree. *ACM Transactions on Algorithms*, 4(1), 2008.

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

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

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

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

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

[8] D. Eppstein and M. T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *ACM GIS '08*, pages 1–10, November, 5-7 2008.

[9] Y. Fang, J. Cao, Y. Peng, and L. Wang. Indexing the past, present and future positions of moving objects on fixed networks. In *CSSE '08*, pages 524–527, Washington, DC, USA, 2008. IEEE Computer Society.

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

[11] T. T. T. Le and B. G. Nickerson. Efficient Search of Moving Objects on a Planar Graph. In *ACM GIS '08*, pages 367–370, Irvine, CA, USA, November, 5-7 2008.

[12] 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.

[13] 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.

[14] Statistics Canada. 2009 road network file. www.statcan.gc.ca, last accessed: September 21, 2009.

[15] Wikipedia. East Los Angeles Interchange. http://en.wikipedia.org/wiki /East_Los_Angeles_Interchange, last accessed: September 21, 2009.