STAR-Tree: An Efficient Self-Adjusting Index for Moving Objects *

Cecilia M. Procopiuc1, Pankaj K. Agarwal2, and Sariel Har-Peled3

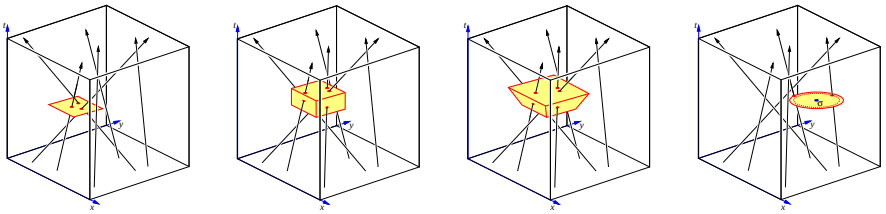
[1] AT&T Research Lab, Florham Park, NJ 07932
magda@cs.duke.edu
[2] Duke University, Durham, NC 27708-0129
pankaj@cs.duke.edu
[3] University of Illinois, Urbana, IL 61801
sariel@cs.uiuc.edu

Abstract. We present a new technique called STAR-tree, based on R*-tree, for indexing a set of moving points so that various queries, including range queries, time-slice queries, and nearest-neighbor queries, can be answered efficiently. A novel feature of the index is that it is self-adjusting in the sense that it re-organizes itself locally whenever its query performance deteriorates. The index provides tradeoffs between storage and query performance and between time spent in updating the index and in answering queries. We present detailed performance studies and compare our methods with the existing ones under a varying type of data sets and queries. Our experiments show that the index proposed here performs considerably better than the previously known ones.

1 Introduction

Motion is ubiquitous in the physical world. Several areas such as digital battlefields, air-traffic control, mobile communication, navigation system, geographic information systems, call for indexing moving objects so that various queries on them can be answered efficiently; see [13, 11] and the references therein. The queries might relate either to the current configuration of objects or to a configuration in the future — in the latter case, we are asking to predict the behavior based on the current information. In the last few years there has been a flurry of activity on extending the capabilities of existing database systems to represent moving-object databases (MOD) and on indexing moving objects; see, e.g., [8, 12, 13]. In this paper we propose a new indexing technique called *spatio-temporal self-adjusting R-tree*, referred to as STAR-tree for brevity, for indexing the trajectories of moving point objects so that range queries and their variants can be

* C.P. is supported by Army Research Office MURI grant DAAH04-96-1-0013 and an NSF grant CCR–9732787. P.A. is supported by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants ITR–333–1050, EIA–9870724, EIA–997287, and CCR–9732787, and by a grant from the U.S.-Israeli Binational Science Foundation.



(a) (b) (c) (d)

Fig. 1: Instances of Q1, Q2, Q2', and Q3 queries, respectively.

answered efficiently. Our technique is a natural extension of R*-tree [6] and is closely related to the index introduced by Šaltenis [11]. Unlike the earlier techniques (e.g. [10, 11]), we neither reconstruct the index periodically nor do we require the index to be updated frequently. Instead, the index maintains certain auxiliary information and updates itself locally to ensure that queries are answered efficiently. The index provides tradeoffs between the storage used and the query performance and between the time spent in updating the index and answering queries.

Section 2 states our model and summarizes the related work and the main contributions of this paper. Section 3 describes the indexing scheme, the events at which the index is updated, and the query and update procedures. Section 4 discusses experimental results. We conclude in Section 5 by summarizing our work and suggesting further directions.

2 Our Contributions and Related Work

Data model Let $S = \{p_1, \ldots, p_n\}$ be a set of n points in $\mathbb{R}^2$, each moving independently. The position of a point p_i at time t is given by $p_i(t) = (x_i(t), y_i(t))$. We use $\bar{p}_i = \bigcup_t (p_i(t), t)$ to denote the *graph* of the trajectory of p_i in the xyt-space. We assume the trajectories to be piecewise-linear functions. The user is allowed to change the trajectory of a point at any time.

Queries The index we propose can be used to answer various types of both *range queries* and *nearest neighbor queries*, based on the current and future positions of the objects. We use the notation *now* to denote the current time. Specifically, we study the following three types of queries.

Q1. Given an axis-aligned rectangle R in the xy-plane and a time value $t_q \geq now$, report all points of S that lie inside R at time t_q, i.e., report $S(t_q) \cap R$; see Figure 1 (a).

Q2. Given a rectangle R and two time values $now \leq t_1 \leq t_2$, report all points of S that lie inside R at any time between t_1 and t_2, i.e., report $\bigcup_{t=t_1}^{t_2} (S(t) \cap R)$; see Figure 1 (b).

Q2'. Given two orthogonal rectangles R_1 and R_2 and two time values $now \leq t_1 \leq t_2$, report all points of S that lie inside $R(t)$ at any time between $t_1 \leq t \leq t_2$, where $R(t) = [(t - t_1)R_2 + (t - t_1)R_1]/(t_2 - t_1)$. That is, report $\bigcup_{t=t_1}^{t_2}(S(t) \cap R(t))$; see Figure 1 (c).

Q3. Given a query point $q \in \mathbb{R}^2$ and a time value $t_q \geq now$, an *ε-approximate k-nearest neighbor query* requires reporting k points $p'_1, \ldots, p'_k \in S$ such that $d(q, p'_i(t_q)) \leq (1 + \varepsilon)d(q, p_i(t_q))$, $1 \leq i \leq k$, where $p_i \in S$, $1 \leq i \leq k$, and $p_1(t_q), \ldots, p_k(t_q)$ are the k nearest neighbors of q. In particular, for $\varepsilon = 0$ and $k = 1$ we obtain the nearest neighbor query over moving points; see Figure 1 (d).

Previous work One direction of research has been to index the trajectories of points, either directly, or by mapping them to higher dimensional points [10, 14]. This is not very efficient since trajectories do not cluster well. Moreover, if they are piecewise linear functions, they are mapped to very high dimensional points, further reducing the efficiency of the index. A second direction is to kinetize static indexing structures. The notion of *kinetic data structures*, introduced by Basch *et al.* [5], has led to several interesting results related to moving objects, including results on kinetic space partition trees (also known as cell trees) [2], and on maintaining the Voronoi diagram of a set of moving objects [9]. In the static case, the Voronoi diagram can be processed into a point-location data structure that efficiently answers nearest-neighbor queries. However, no efficient data structure is known for answering point-location queries in a planar subdivision that is continuously deforming. Agarwal *et al.* [1] developed an index that answers a Q2 query in optimal number of I/Os — provided that the queries arrive in chronological order. Since the index maintains as invariant the sorted order of points along one axis, the number of updates in the worst case is $\Omega(n^2)$. To alleviate this problem, one can parametrize a structure such as the R-tree, which partitions the points but allows the bounding boxes associated with the children of a node to overlap. To expect good query efficiency, the areas of overlap and the areas of the bounding boxes must be small. Maintaining these properties over time is likely to be significantly less expensive than maintaining the stronger invariants that other indexes require. Moreover, the R-treeworks correctly even when the overlap areas are too large, although in this case the query performance deteriorates. A kinetic index called TPR-tree, based on the R*-tree, was proposed by Šaltenis *et al.* [11] to handle range queries over moving points. They parametrize the bounding boxes associated with nodes in the R*-treeas follows. Since it is very expensive to maintain the minimum bounding box at all times, they provide a heuristic so that the coordinate of each vertex is a linear function of time. Although initially this heuristic gives a good approximation of the bounding box, it starts to deteriorate with time. To alleviate the problem, whenever the position of a point p is updated, they recompute the boxes on the nodes along the path to the leaf at which p is stored. Although these frequent updates keep the query time low, they have a large overhead.

Our results We present a *spatio-temporal self-adjusting* R-tree, called STAR for brevity, which is a fully dynamic, R*-tree-based indexing technique. Our approach is similar to TPR-trees, but we introduce the notion of self-adjusting to our index, which allows it to adapt itself without any input from the user, whenever the query performance deteriorates. Our index provides tradeoffs between various performance parameters. A user can specify the parameter that determines the quality of the parametrized bounding box stored at each node of the tree — a better parametrization requires more space. The index also provides a tradeoff between the time spent in self adjusting the structure and the query performance.

3 Indexing Technique

In this section we describe our index STAR-tree. Although it works for points in any dimension, we focus on points in $\mathbb{R}^2$. We first describe the overall structure of the index and the algorithm for computing the parametrized bounding box at each node. We then describe the events at which the index is updated and the procedures for handling the events and for answering queries.

3.1 Overall structure

A STAR-tree is a B-tree $\mathcal{T}$ whose structure is similar to an R-tree. We will use $\mathcal{T}(t)$ to denote the tree at time t. The points are stored in the leaves. More exactly, if the trajectory of a point p_i is a linear function $p_i(t) = a_i + b_i t$, we store the coefficients $a_i, b_i \in \mathbb{R}^2$ at the leaf. For an internal node v, let $S_v(t)$ be the subset of points stored in the leaves of the subtree rooted at v at time t. For each internal node v, we store a pointer to each of its children w and a compact representation of a parametrized bounding box $\mathcal{B}_w(t)$ that contains all the points in $S_w(t)$ for all $t \geq now$. Let $\mathrm{MBB}_w(t)$ be the minimum bounding box containing $S_w(t)$. For each $t \geq now$, the bounding rectangle $\mathcal{B}_w(t)$ contains $\mathrm{MBB}_w(t)$. The quality of the approximation of $\mathcal{B}_w(t)$ to $\mathrm{MBB}_w(t)$ is controlled by a user-specified approximation factor ε. The better the approximation, the larger the space needed to store $\mathcal{B}_w(t)$. Hence the fanout of the tree becomes smaller, since the total size of a node is fixed (e.g., the disk-block size). Our method allows a trade-off between the fanout of the tree and the approximation factor for the minimum bounding boxes of the nodes. We also maintain certain additional information at each node, to be described below, that determines when some of the nodes need to be reorganized.

We fix the initial time to $t_{start} = 0$. We start by bulk-loading a Hilbert R-tree based on the positions of the moving points at t_{start}. We now describe the procedures for maintaining the bounding boxes at each node and for organizing the tree.

3.2 Computing the parametrized bounding box

For each node $v \in \mathcal{T}$, $\mathcal{B}_v$ is parametrized as a piecewise-linear function. That is, we compute a sequence of intervals $\mathcal{I}_v^x(\tau_1^x), \mathcal{I}_v^x(\tau_2^x), \ldots$ along the x-axis, and a sequence of intervals $\mathcal{I}_v^y(\tau_1^y), \mathcal{I}_v^y(\tau_2^y), \ldots$ along the y-axis so that $\mathcal{I}_v^x(\tau_i^x)$ contains the projection of the points in $S_v(\tau_i^x)$ along the x-axis, and $\mathcal{I}_v^x(\tau_j^y)$ contains the projection of the points in $S_v(\tau_j^y)$ along the y-axis, for all i, j. Let $\langle \tau_1, \tau_2, \ldots \rangle$ be the set $\bigcup \tau_i^x \cup \bigcup \tau_j^y$ in sorted order. Then for any $t \in (\tau_i, \tau_{i+1})$ the interval $\mathcal{I}_v^x(t)$ obtained by linear interpolation from $\mathcal{I}_v^x(\tau_i^x)$ and $\mathcal{I}_v^x(\tau_{i+1}^x)$ contains the projection of $S_v(t)$ along the x-axis; a similar result holds for the linearly interpolated interval along the y-axis. Hence, for any t, the box $\mathcal{B}_v(t) = \mathcal{I}_v^x(t) \times \mathcal{I}_v^y(t)$ contains $\mathrm{MBB}_v(t)$. We discuss below how to compute the sequence $\mathcal{I}_v^x(\tau_1^x), \mathcal{I}_v^x(\tau_2^x), \ldots$ so that for each t the interpolated interval $\mathcal{I}_v^x(t)$ is only slightly larger than the minimum length interval that contains the projection of $S_v(t)$ on the x-axis. The computation of the sequence $\mathcal{I}_v^y(\tau_1^y), \mathcal{I}_v^y(\tau_2^y), \ldots$ is similar. This will ensure that $\mathcal{B}_v(t)$ is tight-fitting at any time t. We assume that the set of points $S_v(t)$ remains unchanged for all t. The cases when $S_v(t)$ changes are handled separately in the insertion/deletion and re-balancing procedures.

The problem of computing $\mathcal{I}_v^x(\tau_1^x), \mathcal{I}_v^x(\tau_2^x), \ldots$ is the same as approximating the *extent* of the points in S_v along the x-axis. For simplicity, we describe the algorithm under the assumption that each point is moving with fixed speed, though it works for more general trajectories.

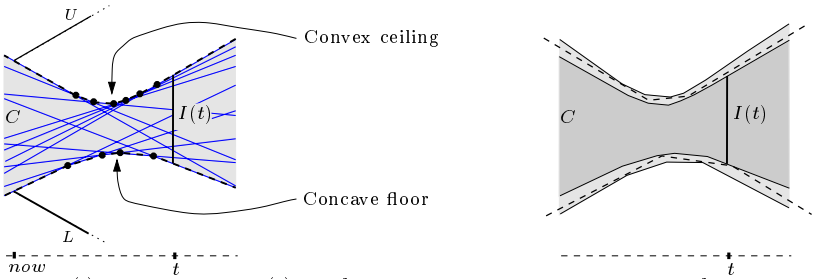


Fig. 2: (i) The interval $I(t)$ is the vertical segment connecting the upper and lower boundaries of C at time t. The rays U, L denote the approximation to $I(t)$ as computed by Jensen *et al.* [11]. (ii) An approximation of C and polygonal chains γ^-, γ^+.

Let $I(t)$ denote the extent of $S_v(t)$ along the x-axis, and let $C = \bigcup_t I(t)$. Then C is a corridor in the tx-plane with convex ceiling and concave floor; see Figure 2. Let ε be the approximation factor. We want to maintain an interval $J(t)$ so that $I(t) \subseteq J(t) \subseteq (1+\varepsilon)I(t)$ for all t. To do this, we compute two polygonal chains γ^-, γ^+ with the following property: Let C' be the corridor formed by $\bigcup_t (1+\varepsilon)I(t)$. The chain γ^- lies between the lower boundaries of C and C', and the chain γ^+ lies between the upper boundaries of C and C', respectively. Moreover, γ^- and γ^+ have only $O(1/\sqrt{\varepsilon})$ vertices. The computation of γ^- and γ^+ is done using the greedy algorithm of [3], which guarantees the

above properties. The approximate extent is defined by $J(t) = [\gamma^-(t), \gamma^+(t)]$; see Figure 2 (ii). The sequence $\tau_1^x, \tau_2^x, \ldots$ is the set of t-coordinates of the vertices of γ^- and γ^+, and $\mathcal{I}_v^x(\tau_i^x) = J(\tau_i^x)$. By contrast, Jensen *et al.* [11] approximate $I(t)$ by two rays (Figure 2 (ii)), and the approximation can become very large over time, decreasing the query efficiency of the index. Our experiments indicate that in practice the number of vertices of γ^-, γ^+ is small (between 3 and 12 for 100,000 points and $\varepsilon = 0.1$). Thus, we can explictly store γ^-, γ^+ at a node of $\mathcal{T}$. If we are unlucky, then γ^-, γ^+ could have too many vertices. In order to ensure that the fanout of a node is not too small, we set a threshold parameter $\lambda \geq 2$ and store only at most λ vertices of each of the four chains. Suppose $t_b(v)$ is the minimum t-value of the last vertex of a chain stored for $\mathcal{B}_v$. Then the parametrization of $\mathcal{B}_v$ is valid only for $t \leq t_b(v)$. We recompute $\mathcal{B}_v(t)$ for $[t_b(v), \infty]$.

If v is an internal node, we do not compute $\mathcal{B}_v$ directly from S_v, but rather in a bottom-up manner from $\mathcal{B}_{w_1}, \ldots, \mathcal{B}_{w_k}$, where $w_1, \ldots, w_k$ are the children of v. There are two issues that one has to handle in this recursive computation. Since $\mathcal{B}_v$ is being computed by approximating the (approximate) bounding boxes computed at it children, the errors accumulate. So one has to choose a smaller value of ε at each node, roughly $\varepsilon/\text{depth}(\mathcal{T})$. Second, the bounding-box representation at the children of v might have been clipped, so $\mathcal{B}_v(t)$ is valid only for $t \leq \min_w t_b(w)$, where w is a child of v. Therefore we set $t_b(v) = \min_w t_b(w)$ if the number of vertices in the chains of $\mathcal{B}_v$ is at most λ. Otherwise, if ξ is the minimum t-coordinate at which a chain of $\mathcal{B}_v$ is clipped then we set $t_b(v) = \min\{\xi, \min_w t_b(w)\}$.

3.3 Events and event queue

There are two types of events — *external* and *internal* that casue the index $\mathcal{T}$ to be updated. External events are insertion/deletion of a point, and change in the trajectory of a point. Due to lack of space, we do not detail the procedures for handling external events. Roughly speaking, insertions and deletions are handled as in the static R-tree, by looking at the snapshot of the index at the time of insertion or deletion. Changing the trajectory of a point usually requires updating the parametrized bounding boxes along the path to the leaf that stores the point. Besides the external events, there are two types of internal events. Recall that at each node $v \in \mathcal{T}$, $\mathcal{B}_v$ is valid only until $t_b(v)$, so if $t_b(v) < \infty$, we have to recompute $\mathcal{B}_v$ at $now = t_b(v)$. We refer to this event as a *box event*. As mentioned earlier, the box events are rare even for small values of ε. Our construction maintains the invariant that $t_b(w) \geq t_b(v)$ for any child w of v.

The second internal event is called a *conflict event*. A conflict event occurs at a node v if the bounding boxes of its children overlap too much. Let $u_1, \ldots u_k$ be the children of v. We replace them with new nodes $w_1, \ldots, w_k$ so that the set of grandchildren of v remains the same and the overlap among the bounding boxes of $w_1, \ldots, w_k$ is smaller than the overlap among those of $u_1, \ldots, u_k$; see Figure 3. That is, we redistribute the grandchildren of v among its children in order to reduce the overlap among the bounding boxes of the children of v.

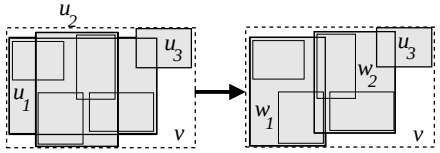


Fig. 3: A conflict event; the children of u_1 nd u_2 are redistributed between w_1, w_2.

There are many ways to define when sibling nodes overlap too much, as well as to compute the new nodes $w_1, \ldots, w_k$. We have experimented with a number of heuristics, all of which try to stay close to the R-tree desideratta that the areas of nodes and the areas of overlaps among sibling nodes should be as small as possible. The problem is hard in our case because the area of $\mathcal{B}_v(t)$ changes with time. For each child w_i of v, we compute the relative area of $\mathcal{B}_{w_i}(t)$ that overlaps with the bounding boxes of its siblings. If a point of $\mathcal{B}_{w_i}(t)$ is covered by multiple boxes, it is counted multiple times. We set an area threshold parameter A_v and compute the overlap for a node only at the times when its area becomes $(1+\alpha_v)A_v$, $(1+2\alpha_v)A_v, \ldots$, $(1+10\alpha_v)A_v$ (α_v is a parameter that we discuss in the experimental section). This approach is related to the intuition that overlaps are only significant for large enough boxes. We set a threshold parameter μ and say that a *conflict* occurs at v at time t if this relative overlap exceeds μ. We set $t_c(v) > now$ to be the earliest time at which a conflict occurs. The value of the area threshold A_v is changed with time because points might move away from each other and thus the areas of bounding boxes increase. We reset A_v every time a conflict event is processed at v. An additional option is to reset this value after each time an external event occurs in a subtree rooted at v. The parameter μ is a ratio between the overlap area and the area of a node. In our experiments, we found that it is best to keep μ unchanged over time.

Finally, we define the *event time* of a node v as $\tau(v) = \min\{t_b(v), t_c(v)\}$. If box and conflict events occur simultaneously, then the box event has higher priority. We maintain a global event queue $\mathcal{Q}$, as a priority queue, that stores the pairs $(\tau(v), e(v))$ for all nodes $v \in \mathcal{T}$ for which $\tau(v) < \infty$. We also use $\mathcal{Q}$ to handle *external events* that are going to occur in the future.

3.4 Answering queries

As mentioned in Section 2 we focus on three types of queries. Since Q1 and Q2 queries are special cases of Q2' queries, we describe the procedure only for the latter. Let R_1, R_2, t_1, t_2 be the given rectangles and time stamps. Let $\mathbf{R}$ be the volume swept by the query rectangle during the interval $[t_1, t_2]$, i.e., $\mathbf{R} = \bigcup_{t=t_1}^{t_2} R(t)$, where $R(t)$ is defined in Section 2; $\mathbf{R}$ is a convex polytope (see Figure 1 (c)). We traverse $\mathcal{T}$ in a top-down manner. At each visited leaf z, we report a point $p \in S_z$ if the line segment $\bigcup_{t=t_1}^{t_2} p(t)$ intersects $\mathbf{R}$. At each internal node v, we recursively visit a child w_i of v if $\mathbf{R}$ intersects $\mathbf{B} = \bigcup_{t=t_1}^{t_2} \mathcal{B}_{w_i}(t)$.

Queries Q3 are processed as follows. Let q be the query point for which we want to report the nearest neighbors at time t_q. We describe an algorithm that

returns the ε-approximate k-nearest neighbors of q, for any $0 \leq \varepsilon$ and $k \geq 1$. For any node v in the STAR-tree, we define the distance between q and v to be $d(q, \mathcal{B}_v(t))$, if q is outside $\mathcal{B}_v(t)$, and 0 otherwise. The query procedure works in two stages. In the first stage it traverses the index in a top-down fashion and at each level i it maintains a set of nodes $\mathcal{L}_i$ as follows. $\mathcal{L}_0$ is the root . For all $i \geq 1$, $\mathcal{L}_i$ is the subset of children of nodes in $\mathcal{L}_{i-1}$ whose distance to q is minimum over all children of nodes in $\mathcal{L}_{i-1}$. Let h denote the leaf level of the STAR-tree, and $\mathcal{L}_h$ be the subset of leaves corresponding to the recursive definition above. We compute the k nearest neighbors of q among all points stored in the leaves of $\mathcal{L}_h$. Assuming that no two nodes on the same level of the STAR-tree overlap and that the minimum side of each bounding box on that level has length $\Omega(\Delta)$, one can prove by a packing argument that the above procedure visits only a constant number of nodes on each level. To describe the second stage, let Δ_k be the distance from q to the kth nearest neighbor candidate computed during the first phase. Starting from the root, we traverse the tree in a depth-first order, using the following criterion. Let v be the current node, so that v is not a leaf. For each child w of v, if $d(q, \mathcal{B}_w(t)) < \Delta_k/(1+\varepsilon)$ then visit w recursively. A fast heuristic is to only execute the first stage. This is expected to report points close to k-nearest neighbors but not guaranteed to do so. In our experiments, the heuristic performed very well (see Section 4.5).

4 Experimental Results

The experiments were performed on a Pentium III 800-MhZ machine with 512MB memory, running Linux. The algorithms are implemented on top of the *Transparent Parallel I/O Programming Environment* [4], a templated library that supports efficient high level implementations of external memory algorithms. The entire data and the index structure reside on disk, with the exception of 200K of cached blocks. The cache implements a LRU replacement policy. The block size is 8192 bytes, which, for two dimensional data, leads to a packing of 227 data points per leaf. By contrast, the TPR-tree uses a block of size 4096 bytes, and the packing parameter is 204 (this is because we store coordinates of type double, and the TPR-tree stores coordinates of type float). Thus, even though we use larger blocks, the leaf fanout and the number of leaves are similar in both structures. Because both the STAR-tree and the TPR-tree have very small height, the upper levels of both structures contain very few nodes, and thus the larger block size in our experiments does not significantly influence the results. Taking into account all these reasons, the performance values obtained with the STAR-tree on the same data and experimental set-up used by the TPR-tree are a good indication on the relative performance of the two indexes.

Parameter choices. In all the experiments $\lambda = 4$, which leads to a fanout of 30 for internal nodes. We set $\varepsilon = 0.1$. The value α_v depends on the level of the node v. The higher the node is (with the root being highest), the larger α_v is. Hence, we check for overlap less frequently at higher levels in the tree, and thus we do

not reorganize upper nodes too often. We have determined experimentally that choosing $\alpha_v = 0.05 \cdot level(v)$ results in a reasonably small number of overlap events, while maintaining good query performance over time. The area A_v is reset after a conflict among children of v is solved, as well as after a child of v is deleted. We do not reset A_v after a child is inserted in v. Each time, we set A_v to be the largest area of a child of v at the time of the reset. The overlap threshold μ is set during the initialization of the node and it does not change over time. In Section 4.1 we evaluate the performance of the STAR-tree for range queries (the only ones for which the TPR-tree performance is reported). We then present experimental results for the nearest neighbor queries in Section 4.5.

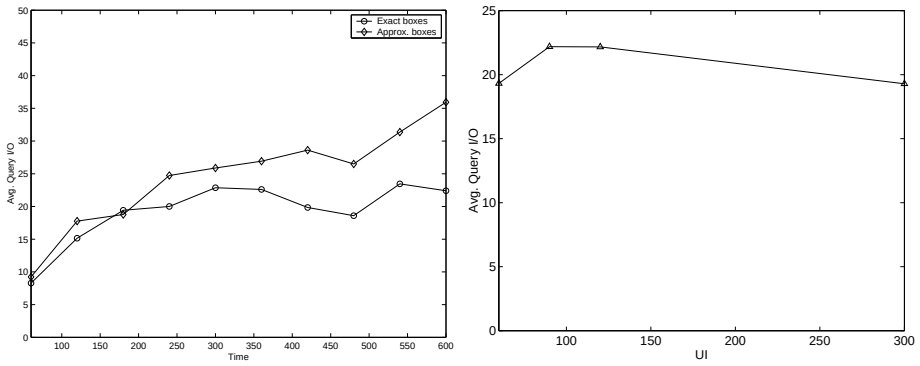
4.1 Range query results

For all experiments reported in this subsection, we perform a combination of queries of types Q1, Q2, and Q2'. The relative proportions of these queries are 0.6, 0.2, and 0.2, the same as in [11]. We use three types of data. The first type was generated using the generator provided by the authors of the TPR-tree, and we use it for comparison purposes. The second type of data sets are similar to the first set in the distribution of initial positions and trajectories, but generate significantly fewer updates, and allow points to appear and disappear from the data. Finally, we provide experimental results on a realistic data set generated by using information on NC roads extracted from the TIGER/Line data of the US Bureau of the Census.

4.2 Comparison with the TPR-tree

We generated $100,000$ points using the data generator provided by the authors of the TPR-tree (see [11] for details). The number of destinations is ND $= 20$. In addition to the initial positions of the points, the generator also outputs a very large number of updates that must be performed by the structure during the course of the simulation. For $100,000$ points, there are close to 1 million point updates, each consisting of two operations: a deletion followed by a re-insertion. Although the updates occur with varying frequencies during the simulation, the average frequency is 7 updates per node per time unit. Hence, the updates act, in effect, as a time-sampling mechanism that allows the indexes to adjust their information very frequently. We ran two experiments: one in which we store the exact bounding box for each node, and one in which we store the approximate box. We report the performance of the STAR-tree in Figure 4 (a). This should be compared to the results reported in [11], Figure 15, on a data set generated with the same parameters. Since we do not have the exact numbers, we chose not plot the performance of the TPR-tree on the same graph. However, we note that, as reported by the authors, the TPR-tree requires an average of about 65 I/O's per query after time $t \geq 360$. Hence, we achieve a speed-up of 3 using exact boxes, and of 2 using approximate boxes. In the case of exact boxes, the overall number of internal events is less than 1.4% of the number of external events (i.e. trajectory updates). When we maintain approximate boxes, there is

only one update event and a similar number of overlap events. We conclude that, with only a small overhead, the STAR-tree is able to adjust itself and maintain better performance over time.



(a) (b)

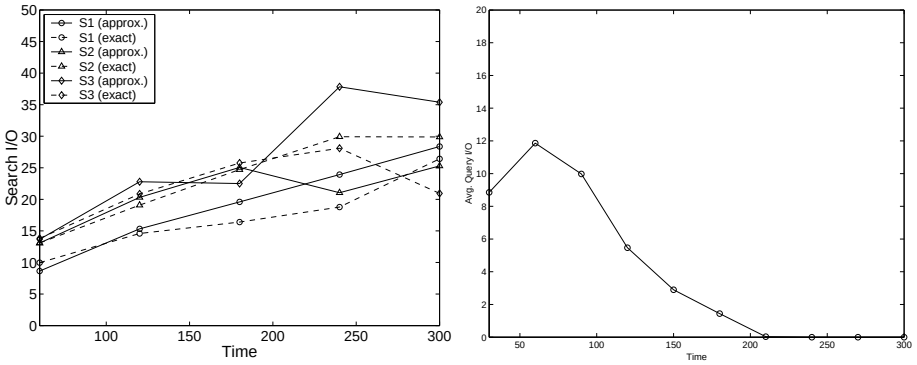
Fig. 4: Performance of STAR-tree under frequent updates: (a) over time; (b) for different UI.

The implementation of the STAR-tree does not require having an estimate on the time horizon for which the tree is in use. This is important for applications on which it is hard to estimate the horizon. For the TPR-tree, the time horizon H is proportional with the update interval UI, i.e. the average time between two succesive updates of the same point. The value H is then used in the bulkloading and updating of the tree. Hence, changing H affects the performance. Figures 9 and 10 in [11] show that for $UI = 60$ the average query I/O is between 40 and 50, depending on the setting of H. For $UI = 120$, the query performance varies between 60 and 90. In contrast, the STAR-tree has the same performance for different settings of UI; see Figure 4 (b).

4.3 Performance of STAR-tree under few updates

In this subsection we study the performance of the STAR-tree when the frequency of updates is significantly lower than for the datasets of the previous section. For example, in applications such as airline traffic, points tend to maintain their velocities and directions for a long period of time. We generate new data sets that closely resemble the previous sets in terms of the distribution of points at loading time, as well as the distribution of trajectories. The significant difference is that no updates are generated while the point travels on a leg between two destinations. We also allow insertions and deletions of the points. We insert 10% of the data points after the initialization, and we delete 10% of the points before the simulation ends. The insertion and deletion times are randomly distributed throughout the simulation. The overall number of external events that the index receives is 11 times smaller than for the previous data set.

The data consists of $100,000$ points. We perform experiments with the number of hubs ND set to 20 (very skewed data), and to $1,000$ (skewed data). We denote by S_1 the first set, and by S_2 the second set. We also denote by S_3 the set of uniform data.



(a) (b)

Fig. 5: Performance of STAR-tree under few updates: (a) skewed and uniform data; (b) realistic Durham data.

Figure 5 (a) shows that the performance of STAR-tree does not deteriorate too much over time. The uniform dataset is the only one for which a significant performance loss appears between $t = 180$ and $t = 240$. However, the index readjusts itself afterwards and the performance improves for $t \geq 240$. For each of S_1, S_2, and S_3, two sets of results are plotted. The first set is for the case when we store the exact bounding boxes, and the second for approximate boxes with $\varepsilon = 0.1$. In the second case, approximate bounding boxes are generated only if storing the chains of the exact boxes requires more than λ vertices per chain. Note that the usage of approximate bounding boxes only marginally worsens the performance for sets S_1 and S_2, and affects the set S_3 only after a significant amount of time. However, the number of box events drops dramatically (e.g., for the set S_1, it drops from 2667 to 9).

The effect of varying the query window size is shown in Figure 6 (i). The average is taken over a simulation time of 300 for set S_1. Finally, Figure 6 (ii) shows the influence of data size on the average query performance. All sets we use are uniform, because this is the case in which the index has worst performance. The averages are for a simulation time of 300.

4.4 Realistic traffic data

We have generated $51,000$ data points that model car traffic as follows: We extracted the roads map around Durham, NC (within a square of 120 miles centered at Durham) from the National Transportation Atlas Data provided by Bureau of Transportation Statistics (see `http://www.bts.gov/gis/ntatlas/natlas.html`).

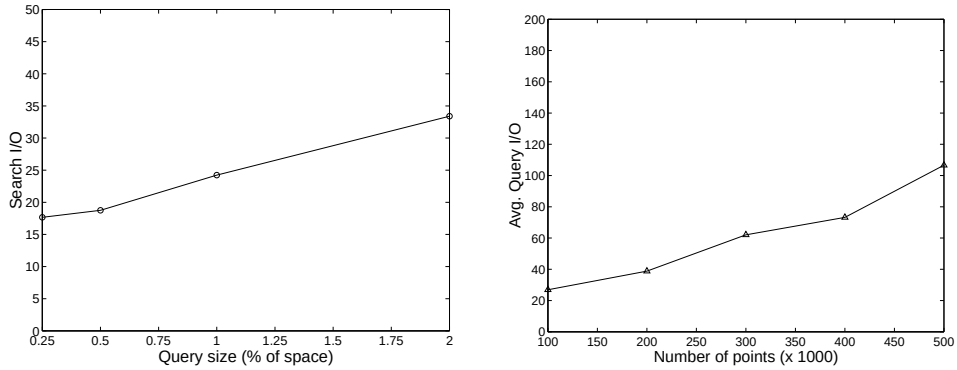


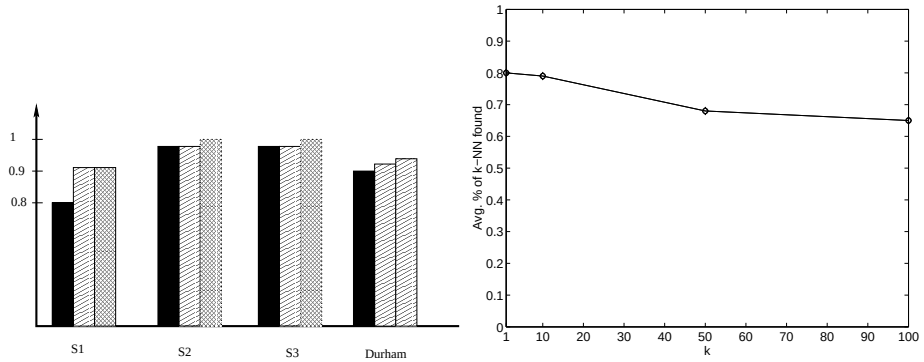
Fig. 6: (i) Effect of query size on the search performance of the STAR-tree. (ii) Search performance of the STAR-tree for varying number of points.

This resulted in a a set of $231,554$ polygonal chains describing the roads around Durham. Using this information we computed a planar-map representing the road map of the region. To generate a realistic path, we randomly selected two points on the map, and computed (using Dijkstra's algorithm) the shortest path on the map between those points. We then simplified each path using the Douglas-Peucker heuristic [7] so that the simplified path has at most five vertices (to avoid frequent trajectory updates). The simplified path is typically almost identical to the original path. The routes are generated in three length ranges (relative to the diameter of the graph). A car is then generated to travel along the route, and its speed varies according to the length of the route. When a car reaches the final point on its route, it signals the index that it should be deleted. Figure 5 (b) shows how the performance of the search changes over time. The experiment was run using approximate bounding boxes with $\varepsilon = 0.1$. Because of the cache, the average I/O for the last 60 time units is 0.

4.5 Nearest neighbor query results

During each time unit, we generate four nearest neighbor queries uniformly at random over the entire data space. Unless otherwise specified, each simulation lasts 200 time units. The data sets are the same from the previous section.

Accuracy results We analyze the accuracy with which the heuristic method answers nearest neighbor queries using various measures. In Figure 7 (a) we show results for 1-nearest neighbor queries. The solid bars show the percentage of queries for which our procedure returns the exact 1-nearest neighbor. The hashed bars represent the percentage of queries for which the nearest neighbor as computed by our procedure is a 0.1-approximate nearest neighbor. Finally, the grid-patterned bars represent the percentage of queries for which the nearest neighbor computed by our procedure is one of the exact 10-nearest neighbors. For sets $S2$ and $S3$, these percentages are in the range of $98-99\%$, while for $S1$ and $Durham$ they are lower, ranging from 80% to 96%.



(a) (b)

Fig. 7: Accuracy of heuristic method for: (a) 1-nearest neighbor; (b) k nearest-neighbor.

For each data set, we also compute the accuracy of answering k-nearest neighbor queries, for $k = 10$. For each query, we determine how many of the 10 points we return are among the exact 10-nearest neighbors of the query, and we average this number over all queries. Our experiments show that for $S2$ and $S3$ this average is about 9.8, implying that for almost all queries we return the exact 10-nearest neighbors. For $S1$, the average is 7.9, and for $Durham$ it is 8.7. However, the accuracy in terms of the average percentage of exact k nearest neighbors found degrades as k increases. Figure 7 (b) illustrates this behavior for set $S1$. We report the average (over all queries) percentages of exact and 0.1-approximate nearest neighbors that the heuristic finds.

Efficiency results We study the efficiency of the exact, approximate and heuristic methods in terms of the average number of I/O's required to answer a query. The heuristic method is always faster, since we only execute one traversal of the index (see the previous section). The exact method is at most as fast as the approximate method. We report our results both in the presence and in the absence of the cache. As we show below, the presence of the cache causes all three methods to have similar efficiency. This is to be expected, since some nodes visited during the second traversal are likely to have been cached during the first traversal.

We have run experiments for $k = 1$, 5, and 10, and for $\varepsilon = 0$, 0.1, 0.2, and 0.4, and have concluded that the average number of I/O's does not change significantly for these values of k and ε. This indicates that the 10 nearest neighbors of most queries are stored together. Moreover, in the absence of the cache, our experiments show that the average number of I/O's for a query is not much larger than the depth of the tree (for the heuristic method), or twice the depth of the tree (for the other two methods). This is an indirect indication that the STAR-tree has small overlap among nodes on the same level. Figure 8 shows experimental results for $k = 10$ for sets S_1, S_2 and S_3 both in the presence and in the absence of the cache, for the heuristic (dashed lines) and exact methods

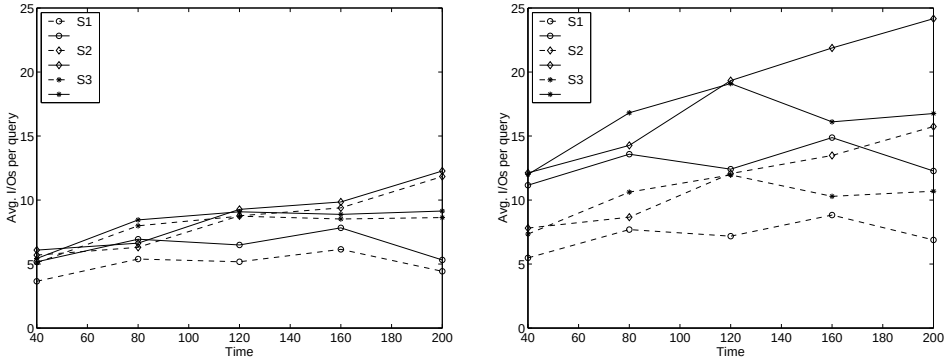


Fig. 8: I/O efficiency for (a) cache present; (b) cache not present.

(solid lines). Results for the approximate method (with $\varepsilon = 0.1$) are close to those for the exact method. We report the average query I/O every 40 time units. We expect that the efficiency of the exact and approximate methods will decline when k becomes too large. The number of I/O's in the heuristic method does not depend on k. Hence, its efficiency will remain the same. However, its accuracy in terms of the number of exact k nearest neighbors found will degrade. Due to lack of space, we do not report experiments for the realistic data set (since this set is smaller, it should not be reported together with the other sets). However, these experiments show that the index adapts itself as points are deleted, and the query performance improves after 100 time units.

5 Conclusions

We presented a new technique called STAR-tree for indexing moving points in the plane. Our experiments clearly demonstrate the advantages of our technique. By intergrating a few geometric techniques with the indexing techniques, we were able to circumvent the problem of updating the index frequently and to improve the query performance. The next step would be to handle uncertainty, to answer more complex queries, and to incorporate a few learning techniques in the self-adjusting procedure.

Acknowledgments

The authors thank Jeff Erickson, Christian Jensen, and Ouri Wolfson for various useful discussions, Simonas Šaltenis for answering several questions related to his paper [11], Simonas and Christian for providing their data generator, and Jan Vahrenhold for providing his R-tree code.

References

1. P. K. Agarwal, L. Arge, and J. Erickson, Indexing moving points, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 2000. 175–186.
2. P. K. Agarwal, J. Erickson, and L. J. Guibas, Kinetic BSPs for intersecting segments and disjoint triangles, *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, 1998, pp. 107–116.
3. P. K. Agarwal and S. Har-Peled, Maintaining approximate extent measures of moving points, *Proc. 12th ACM-SIAM Sympos. Discrete Algorithms*, 2001.
4. L. Arge, R. Barve, D. Hutchinson, O. Procopiuc, L. Toma, D. E. Vengroff, and R. Wickeremesinghe, *TPIE User Manual and Reference (edition 0.9.01b)*. Duke University, 1999. The manual and software distribution are available on the web at `http://www.cs.duke.edu/TPIE/`.
5. J. Basch, L. J. Guibas, and J. Hershberger, Data structures for mobile data, *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, 1997, pp. 747–756.
6. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, The R*-tree: An efficient and robust access method for points and rectangles, *Proc. ACM SIGMOD Conf. on Management of Data*, 1990, pp. 322–331.
7. D. H. Douglas and T. K. Peucker, Algorithms for the reduction of the number of points required to represent a digitized line or its caricature, *Canadian Cartographer*, 10 (1973), pp. 112–122.
8. R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis, A foundation for representing and querying moving objects, *ACM Trans. Database Systems*, 25 (2000), pp. 1–42.
9. M. I. Karavelas and L. J. Guibas, Static and kinetic geometric spanners with applications, *Proc. 12th Annu. ACM-SIAM Sympos. Discrete Algorithms*, 2001, pp. 168–176.
10. G. Kollios, D. Gunopulos, and V. J. Tsotras, On indexing mobile objects, *Proc. Annu. ACM Sympos. Principles Database Syst.*, 1999, pp. 261–272.
11. S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez, Indexing the positions of continuously moving objects, *Proc. ACM SIGMOD International Conference on Management of Data*, 2000, pp. 331–342.
12. A. P. Sistla and O. Wolfson, Temporal conditions and integrity constraints in active database systems, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, 1995, pp. 269–280.
13. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao, Modeling and querying moving objects, *Proc. Intl Conf. Data Engineering*, 1997, pp. 422–432.
14. J. Tayeb, O. Ulusoy, and O. Wolfson, A quadtree-based dynamic attribute indexing method, *The Computer Journal*, (1998), 185–200.