

Indexing the positions of continuously moving objects

Abstract

This report is based on the paper (Saltenis et al., 2000).

In the current world of wireless communication, the technical possibilities of recording the position of wireless device have strongly increased. The continuous tracking of moving objects is a special challenge. This report shows a new tracking technique which records such moving objects in order to query their current and future position. The technique builds on an R*-tree and is able to index objects in one-, two- and three dimensions. Update algorithms which allow the user to remove objects or to change their anticipated position in the future.

1 Introduction

The positioning of objects has grown considerably in recent years thanks to developments in technology such as GPS, WLAN and other wireless systems.

Continuously moving objects represent a new difficulty for classical database technology. Earlier saved data was thought to remain unchanged in the database until explicit changes of the data appeared. Storing objects constantly in movement would result in a non-stop update of positions, which would overburden the database. The other possibility is to store the objects position very rarely. But the disadvantage of this possibility is that most of the time the stored positions of the objects would not be correct. Therefore both possibilities are not practical, and a new technique has to be used.

The new technique doesn't only save the actual position of an object, but also its current speed vector. This technique excludes the use of explicit updates if the object moves at constant speed (O. Wolfson et al., 1998). With this new technique the position of the object is considered as the result of a function whose input data are object position at a given reference time and the current time. So knowing the object's position at reference time and the time would result in knowing the position at this current time. Of course an object is able to change its speed or direction. In this case an update of the new data is needed.

The following introduced indexing technique uses, a time parametrized R-tree called TPR-tree, which is specialized for saving moving objects current and future positions. It is based on the R*-tree (N. Beckmann et al., 1990).

The bounding rectangles in the TPR-tree are also functions of time just like the moving points they enclose.

The following section two explains how the data has to be indexed, the supported queries and the problem parameters. Section 3 explains the TPR structure and algorithms. Section 4 summarizes the performance experiments and section 5 summarizes all sections and shows

additional research possibilities.

2 Problem Statement and Related Work

The indexing technique as well as the supported queries and the problem parameters are explained.

2.1 Problem setting

The position of an object at time t is defined by $x(t) = (x_1(t), x_2(t), \dots, x_d(t))$, assuming that t is not before the current time. The idea is to take those future time positions and model them into a linear function of time. This function needs to be defined by two variables. One is the position of the object at a specific time $x(t_{ref})$ and the other is the velocity vector of the object having d -dimensions $v = (v_1, v_2, \dots, v_d)$. Those two functions result in the following linear function of time $x(t) = x(t_{ref}) + v * (t - t_{ref})$. The first parameter $x(t_{ref})$ can be the actual position of the object or a position that the object is going to have at a chosen reference time.

This idea of using a linear function allows saving many database recording actions, since only meanderings, that would not fit into the saved linear function have to be recorded. Objects then only have to report a change, when they have changed their path. Not when they just have changed their position (O. Wolfson et al., 1999; D.Pfoser et al., 1999; Moreira et al., 2001).

The reference position and the velocity vectors will also be used to record the future coordinates of the bounding rectangles.

In order to illustrate the mentioned facts figure 1 contains on the top left diagram 7 objects positions and speed vectors.

Creating an R-tree at time $t = 0$ would result in minimal bounding rectangles at time $t = 0$, which you can see on the top right diagram.

Unfortunately when time goes by, the rectangles grow in this example. And the query performance gets more and more worse. This can be seen in the bottom left diagram. This decline is due to different speed vectors.

Considering this fact it would have been better to structure the objects as in the bottom right diagram at $t = 3$. However this would have lead into slower query performance at $t = 0$. This fact

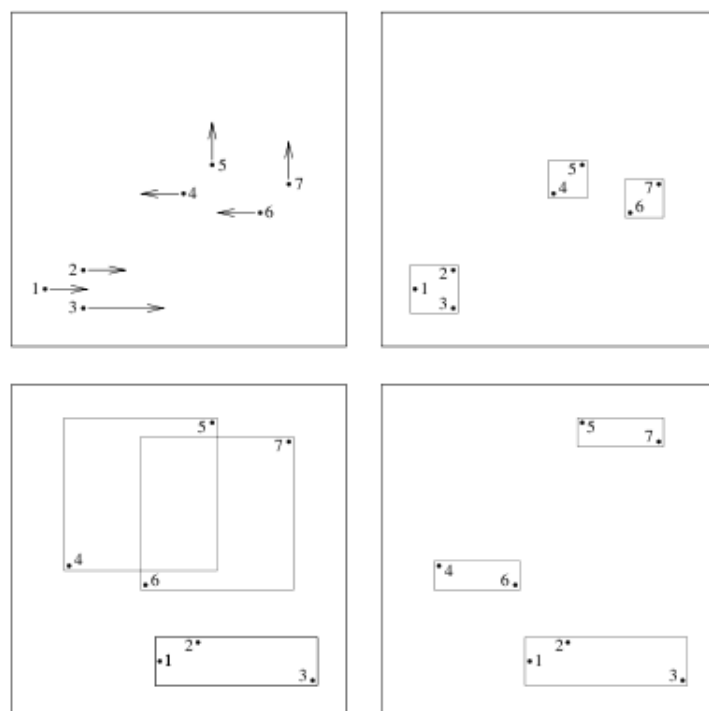


Figure 1: Moving Points and Resulting Leaf-Level MBRs

shows that the MBR- Algorithm should consider when the most queries would be posed.

The algorithm presented in section 3 takes those factors into account.

2.2 Query Types

In the following section all possible query types are going to be summarized and categorized. First a d -dimensional rectangle R is specified by its d projections $[a/1, a\backslash1], \dots [a/d, a\backslash d]$. A/d means for example variable a in the dimension d with / the lower boundary, which is delimited by the upper boundary $a\backslash d$ in the dimension d . Now let R, R_1, R_2 be three different d -dimensional rectangles and $t, t_1 < t_2$, be three different time values all equal or more recent than the current time.

Type 1 time slice query: $Q = (R, t)$ is a rectangle R at time point t .

Type 2 window query: $Q = (R, t_S, t_E)$ is for a rectangle over the interval $[t_S, t_E]$. This query goes through the (x, t) -hyperspace crossing the $(d+1)$ -dimensional rectangle $([a_{S1}, a_{E1}], [a_{S2}, a_{E2}], \dots, [a_{Sd}, a_{Ed}], [t_S, t_E])$.

Type 3 moving query: $Q = (R_1, R_2, t_S, t_E)$ defining a $(d+1)$ -dimensional trapezoid containing R_1 at t_S with R_2 at t_E .

The first query is the simplest one. It is a special case of the second, and the second one itself is also a special case of the third one.

Figure 2 shows an example of all three different queries in a one dimensional world.

Timeslice queries are Q_0 and Q_1 . They are looking for an area at a specific time.

Q_2 is a window query since an area is examined over a time period.

Q_3 is a moving query since a varying area is examined over a period of time.

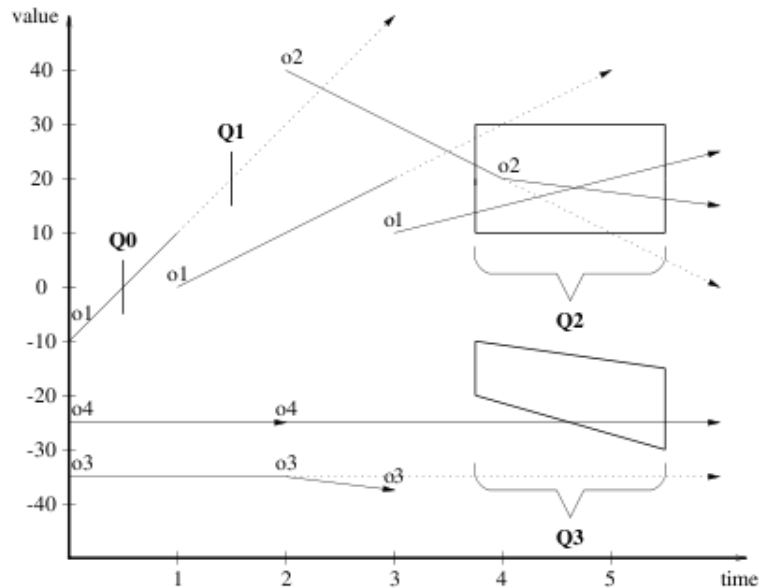


Figure 2: Query Examples for One-Dimensional Data

We define $iss(Q)$ as the time when the query Q is issued. The parameters of an object, namely its reference position and the velocity vector depend on $iss(Q)$, because they often can change over time. We now define an object o_1 whose speed vector has three different forms depending on whether $iss(Q) < 1$, $1 \leq iss(Q) < 3$ or $3 \leq iss(Q)$. a query q_1 would then lead to an answer o_1 if $iss(Q_1) < 1$ and no object would result if $1 \leq iss(Q_1)$.

This shows that a query gets less accurate when going in the future. Knowing this fact implicates that real world applications should not go too far into the future.

2.3 Problem parameters

To describe the quality of a TPR-tree, there are several Problem parameters:

-Querying window (W): W describes how much a query can read into the future. For type 1 queries it is defined as $iss(Q) \leq t \leq iss(Q) + W$, and for type 2 and 3 queries it is defined as $iss(Q) \leq tS \leq tE \leq iss(Q) + W$.

-Index usage time (U): U describes the time for which an index can be used for querying. $tl \leq iss(Q) \leq tl + U$, where tl stands for the moment when the index was created or loaded.

-Time horizon (H): The time interval t, tl and $t2$, during which queries are drawn. Basically it is essential that $H = U + W$.

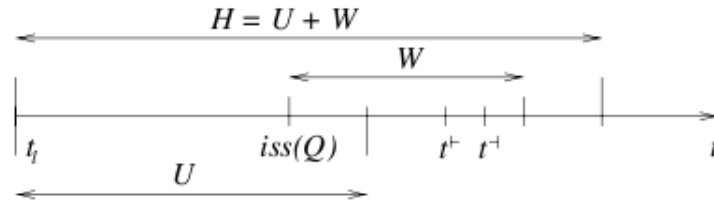


Figure 3: Time Horizon H , Index Usage Time U , and Querying Window W

Imagining that an index would be used at the end of U and would look U time units into the future, results in the fact that an index which was created at tl should support all queries which are H time units in the future.

3 Structure and algorithms

The algorithms and the structure of an TPR-tree are now going to be explained. There is also an explanation about how the time-parametrized bounding rectangle is defined. In the following section there are also dynamic update algorithms and how the dynamic update algorithms tailor the tree to the time horizon H . In this section we use the term bounding interval for a one-dimensional bounding rectangle and bounding rectangle for any d -dimensional hyper-rectangle.

3.1 Index Structure

The TPR-tree has a similar structure as an R-tree. The entries in the leaf nodes are the position of the moving point and a pointer to the moving point. The entries in an internal node are combinations of a pointer to a subtree and a rectangle, which encases the moving points or the bounding rectangles of the subtree.

The position of a moving point and its velocity vector- (x,v) represent a moving point in one dimension, if $x = x(t_{ref})$. We assume that t_{ref} is equal to the index creation time tl .

In order to bound the moving points, the bounding rectangles are also time parametrized.

This means that their coordinates are also functions of time just like the moving points.

In general there is a difference between the tightness of bounding the rectangle and the storage cost of saving those bounding rectangles. Using time-parametrized bounding rectangles which are always minimal results into enclosing all moving points on its own. They can be seen in Figure 4.

The better way in place of using minimum bounding rectangles is the way that the TPR-tree uses. The TPR-tree uses conservative bounding rectangles; which are minimal at some time; but not always. The way how they work is illustrated in Figure 4. First in $t = 0$ a minimal bounding rectangle is defined, enclosing both moving points A and B. At $t = 3$ the rectangle has grown.

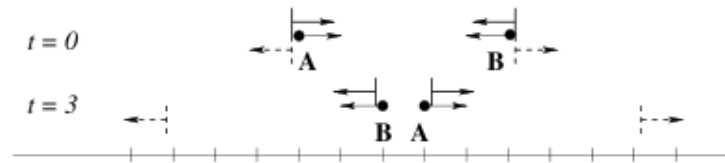


Figure 4: Conservative (Dashed) Versus Always Minimum (Solid) Bounding Intervals

Speed into the left direction is negative and speed into the right direction is positive. The leftmost point of the rectangle at $t = 0$ has the smallest speed, which in this case is the one from B. And the rightmost point of the rectangle has the biggest speed, which in this case is A's speed. Doing so ensures that certainly at any time all points are enclosed within the rectangle after $t = 0$. Conservative bounding rectangles never shrink, but if all points move in the same direction they would have a constant size.

The above mentioned conservative bounding rectangle is now summarized in a formal way. Assuming $t_{ref} = t_l$ a time-parametrized bounding intervals defined as $[xS(t), xE(t)] = [xS(t) + vS \cdot (t - t_l), xE(t_l) + vE \cdot (t - t_l)]$ as (xS, xE, vS, vE) whereas

$$xS = xS(t_l) = \min_i \{oi.xS(t_l)\}$$

$$xE = xE(t_l) = \max_i \{oi.xE(t_l)\}$$

$$vS = \min_i \{oi.vS\}$$

$$vE = \max_i \{oi.vE\}$$

In this definition, oi stands for the range to be enclosed. $\min_i \{oi.vS\}$ means for example the lowest speed of all elements in the range oi . If only one point is considered and not an interval with many points, then $oi.xS(t_l)$ and $oi.xE(t_l)$ would change into $oi.x(t_l)$ and $oi.vS$. Therefore $oi.vE$ can be summarized as $oi.v$ since there would be only one point remaining.

The problem is that such minimum bounding rectangles in the best case persist in their size and in the other cases grow. Due to this fact an update of the boundaries of the rectangle is quite useful. In order to update the rectangle there are two useful formulas to compute the new boundary, whereby t_{upd} is equal to the time of the update:

$$xS = \min_i \{oi.xS(tupd)\} - vS(tupd - tl)$$

$$xE = \max_i \{oi.xE(tupd)\} - vE(tupd - tl)$$

An example of this mechanism can be seen in figure 5.

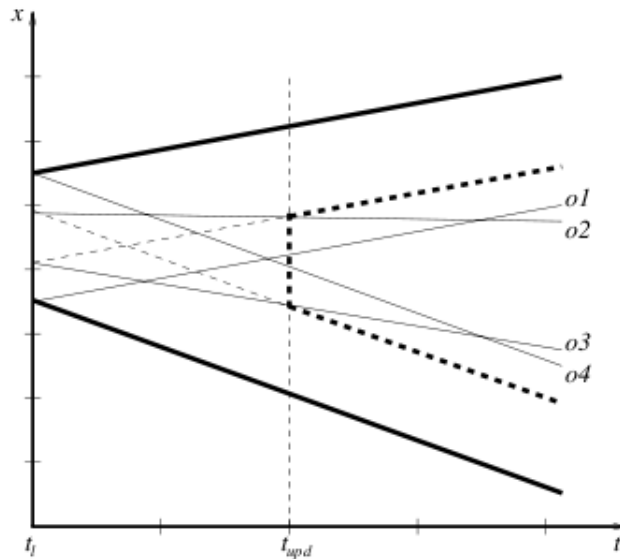


Figure 5: Load-Time (Bold) and Update-Time (Dashed) Bounding Intervals for Four Moving Points

The bold lines are the rectangles at $t = 0$. The dashed lines are the rectangles at the update time. In this example it is clear to see that an optimization can not be done indefinitely when the objects continue their ways and that the bold and the dashed line are parallel, since their speed vectors were not changed.

3.2 Querying

A time slice query proceeds the TPR-tree just like an R-tree, with the small difference that the bounding rectangles need to be calculated for the time of the query. A bounding interval formally expressed with (xS, xE, vS, vE) only satisfies a query $(([aS, aE]), tq)$ if $aS \leq xE + vE * (tq - tl)$ and $aE \geq xS + vS * (tq - tl)$.

The answer to window or moving queries is a bit more difficult. A graphical illustration of the problem can be seen in figure 6.

The formal definition of the is a time

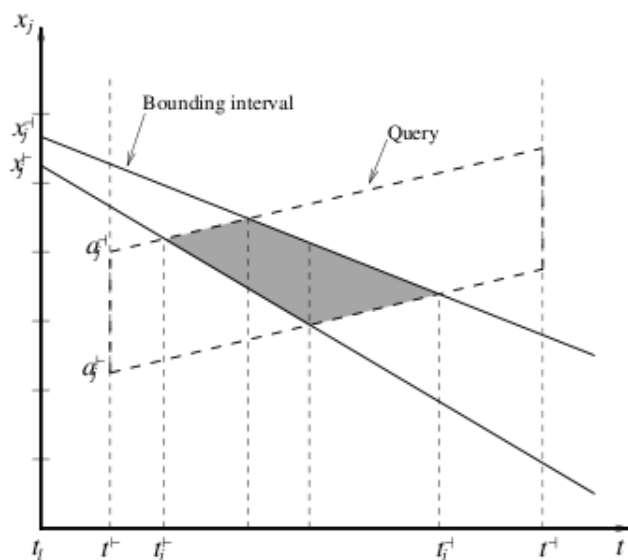


Figure 6: Intersection of a Bounding Interval and a Query

parametrized rectangle R given the parameters ($aS_1, aE_1, aS_2, aE_2, \dots, xS_d, xE_d, vS_1, vE_1, vS_2, vE_2, \dots, vS_d, vE_d$) intersects the moving query $Q = ([aS_1, aE_1], [aS_2, aE_2], \dots, [aS_d, aE_d], [wS_1, wE_1], [wS_2, wE_2], \dots, [wS_d, wE_d]), tS, tE$. The w letters are in this case velocity vectors obtained from subtracting R_2 from R_1 .

The algorithm calculates for every dimension the time interval of overlapping in this specific dimension. Having the time intervals $I_j = [tS_j, tE_j]$ for every dimension j ($j = 1, 2, \dots, d$), the intersection $[tS, tE]$ of those intervals is computed. This intersection $[tS, tE]$ contains the only interval, where objects overlap in every dimension.

The algorithm uses the following formulas to compute I_j :

I_j is null **if** $aS_j > xE_j(tS)$ and $aS_j(tE) > xE_j(tE)$
or $aE_j < xS_j(tS)$ and $aE_j(tE) < xS_j(tE)$
otherwise I_j is equal to $[tS_j, tE_j]$

The first expression ($aS_j > xE_j(tS)$ and $aS_j(tE) > xE_j(tE)$) means that Q is above R and the second expression means that Q is below R . The formulas for tS_j and tE_j are as following,

tS_j is

$tS + (xE_j(tS) - aS_j)/(wS_j - vE_j)$ **if** $aS_j > xE_j(tS)$
 $tS + (xS_j(tS) - aE_j)/(wE_j - vS_j)$ **if** $aE_j < xS_j(tS)$
 tS **otherwise**

Again the first line states that Q is above R at tS , and the second predicates that Q is below R at tS . The formulas for tE_j are:

tE_j is

$tS + (xE_j(tS) - aS_j)/(wS_j - vE_j)$ **if** $aS_j(tE) > xE_j(tE)$
 $tS + (xS_j(tS) - aE_j)/(wE_j - vS_j)$ **if** $aE_j(tE) < xS_j(tE)$
 tE **otherwise**

Again the first line declares that Q is above R and the second one that Q is below R at tE .

3.3 Heuristics for tree Organisation

The mechanism of the insertion algorithm in function of the time horizon H is explained in

this section.

This will help to understand section 3.4 about the insertion algorithm. If H is not big at all, then the tree could use existing R-tree insertion algorithms. This is due to the fact that only the initial positions really matter, the movement of the objects and the growth of the bounding rectangles do not matter, since only very near-term queries are supported. But when H is large it is quite important to use an insertion algorithm which considers the velocity vectors of the moving points. This can be done by considering that the growth rates of the bounding rectangles should be small at any time.

Those thoughts lead to the following approach. The R*-tree insertion and bulk loading algorithms, minimize functions as the areas of the bounding rectangles, their perimeters, and the overlap among the bounding rectangles. In our example all those functions are time depended. Since for example a minimal bounding rectangle can have a bigger area at later times. So the the functions start from t_l and end in $t_l + H$. This leads to the idea to use an integral that can be minimized.

$$\int_{t_l}^{t_l+H} A(t)dt \quad (1)$$

In figure 6 this integral would be the area between the lines of $x \setminus j$ and x/j . The explanation of the use of this formula for dynamic update algorithms follows in the next section.

3.4 Insertion and deletion

An R*-tree uses algorithms that calculate the area of a bounding rectangle, the intersection of two bounding rectangles, the margin of bounding rectangle, and the distance between the centers of two bounding rectangles (Beckmann et al., 1990). The insertion algorithm of the TPR-tree works similarly with the little difference that as mentioned in the previous section integrals of those functions over $[t_l, t_l+h]$ are used.

Previously the three parameters $H = U + W$ were defined. If updates are quite frequent H should not be much bigger than W . Nevertheless if updates are rare H can exceed W a little bit more.

Using integrals are very important in order to describe the difference between an R*-tree and a TPR-tree. Another important difference is the splitting algorithm. While the R*-tree split bases its decision on the current positions along each of the coordinate axes, the TPR-tree split algorithm considers the different positions of the objects over time. With load-time bounding rectangles, positions at t_l are used. With update-time bounding rectangles positions at the current time are used.

The split algorithm also ponders on the velocity vector of the moving points, summarizing similar velocity vectors more often than non similar velocity vectors.

The deletions are done as in the R*-tree. And if a node has not enough points, it is eliminated and its entries are reinserted.

3.5 Comparison of the TPR to the R-tree

The TPR-tree is in opposite to the static R-tree a dynamic data structure since the boundaries change all the time. The TPR-tree has also great advantages with moving objects compared to the R-tree which always needs to update the positions. This is due to the fact that the TPR-tree uses functions in order to represent the objects current and future positions. The R-tree in contrast does only save the objects current positions. Having saved the objects speed at a reference time allows the TPR-tree to use sophisticated query algorithms which the R-tree does not have. The result can be seen in the experiment section.

4 Performance experiments

This section contains performance tests which compare the TPR-tree with other Data-structures in order to give an impression about its productivity.

4.1 Experimental setup and workload generation

The experiments were done with the Generalized Search Tree Package, GiST (Hellerstein et al., 1995). The performance studies use workloads with alternating queries and update operations on the index. This should simulate the normal usage of the trees.

The simulated space was in a 1000 * 1000 kilometers area. There are 20 destinations to which the 100'000 moving points randomly move to. There are 3 different possible maximal speeds for the objects 0.75, 1.5 and 3 km/min (45, 90 and 180 km/h). During the first sixth of a route the objects accelerate from zero to their speed, in the middle two thirds they remain at constant speed and the last sixth they slow down from their speed to zero. Once arrived to the destination they randomly choose a new one. The Update constant UI is usually set to 60, which means that in average 60 time periods are between two updates.

The above described workloads did run for 600 time units. This results in a total of one million update operations. Additionally there are also queries to support the update algorithm. A total of 2400 queries has been done, approximately 1440 from them are time slice, 480 are window and 480 are moving queries. The spatial part of the query is in most experiments equal to 0.25 %.

4.2 Investigating the Insertion Algorithm

Figure 7 shows the search performance for an TPR-tree with $UI = 60$. The three different curves represent three different querying windows. On the figure it can clearly be seen that the best values of H are between $UI/2 + W$ and $UI + W$, since the search I/Os are then at the lowest point. In the following experiments H is though set equal to $UI/2 + W$.

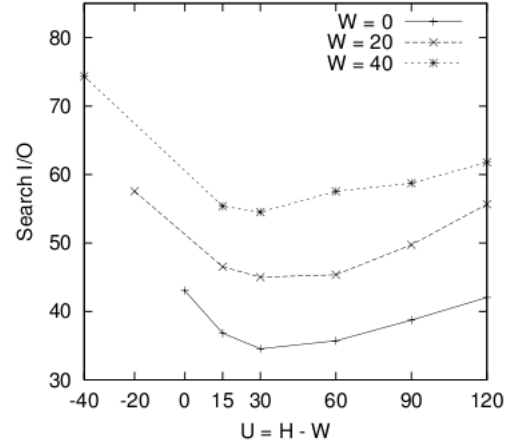


Figure 7: Search Performance For $UI = 60$ and Varying Settings of H

4.3 Comparing the TPR-tree to its alternatives

In Figure 8 the query performance is shown for a varying number of destinations. The statistic clearly shows that TPR-tree with updated Minimal Bounding Rectangles outperforms the other three solutions. One is the classical R-tree and the other one is a TPR-tree without updated Minimal Bounding Rectangles. Having only two destinations is much easier since most of the moving points have similar speed vectors.

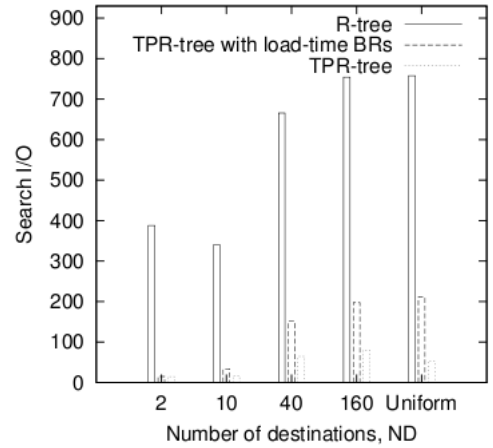


Figure 8: Search Performance For Varying Numbers of Destinations and Uniform Data

Although a TPR-tree with updates handles search queries better than without, it must be also considered that an update itself causes some cost. As an example, a workload with 10 destinations would have 17 I/Os instead of 33 having used update costs 1.6 I/Os instead of 1.3. This clearly shows that the update is worth it.

Figure 9 inspects the impact of different querying windows, namely for different values of W on the querying performance. It is shown that the querying window itself does not have a great impact at all. But again the R-tree gets clearly outperformed by the two implementations of the TPR-tree. It is also shown that adjusting the Minimal Bounding Rectangles results to an increase of performance as the TPR-tree uses less I/Os than the TPR-tree with load time BRs.

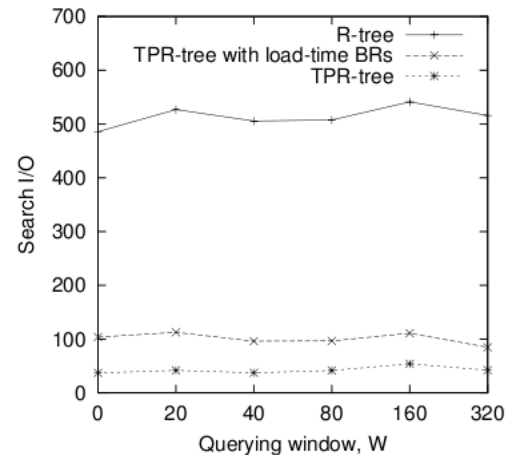


Figure 9: Search Performance for Varying W

Figure 10 shows the search performance for various values of the query size, which is an indication of the size of the total space the queries cover. The R-tree with reduced H uses a value of 120. This can be done since the data which was created does not use update interval longer than $2 \cdot UI$ and UI is equal to 60 in the experiments. The R-tree then significantly improves though its performance is still weaker than the one of the TPR-tree.

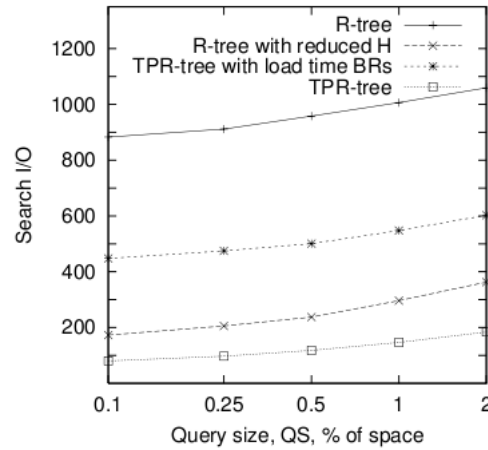


Figure 10: Search Performance For Varying Query Sizes and Three-Dimensional Data

Figure 11 illustrates the search I/Os depending on the number of objects. The R-tree is not on the diagram since it uses far too much operations in order to find the objects.

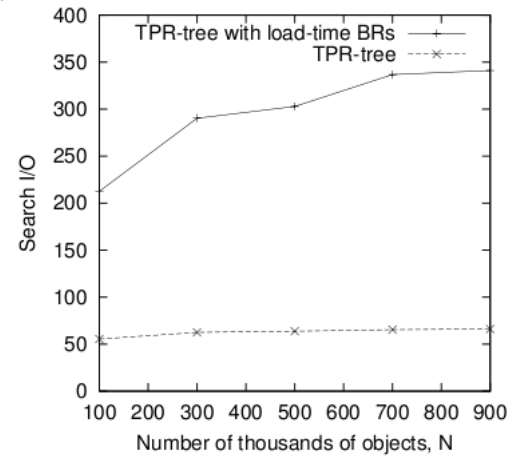


Figure 11: Search Performance for Varying Number of Objects

Figure 12 shows the performance of the trees depending on the time which actually passed. The performance of the TPR-tree diminishes until time 360 but then it remains constant. This is due to the fact that at the beginning node splits often occur, dynamic updates are performed and the bounding rectangles change. But after some time the tree stabilizes. The performance of the TPR decreases constantly since the bounding rectangles get larger and larger.

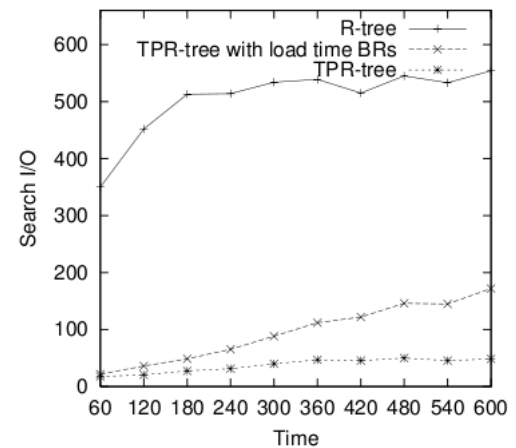


Figure 12: Degradation of Search Performance with Time

5 Concrete examples

This section shows some concrete applied examples. First the insertion is explained, then the concept of the minimum bounding rectangles, the querying and the deletion.

5.1 Insertion

The insertion starts with the situation in Figure 13. Two moving objects O1 and O2 are assigned both in one branch of the TPR-tree. Both objects move over the time t in the region of x . This diagram therefore only has one dimension.

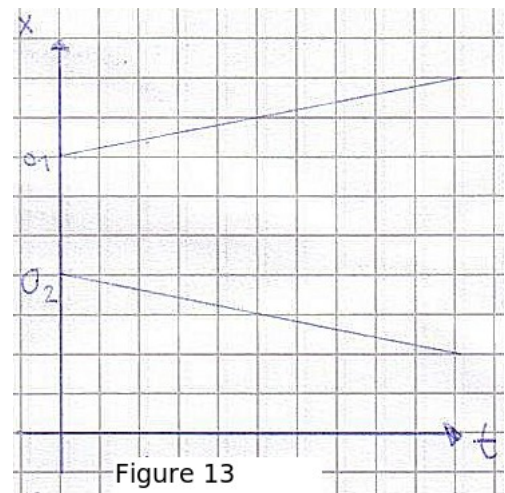


Figure 13

In Figure 14 an additional object appears. It is called O3. Because the branch is now overfilled a split operation has to be done to separate the objects. The split operation assigns O1 and O3 in a branch and O2 in another one. This is because over the time the area and the extend of a O1/O3 are smaller than the area of O2/O3. The insertion algorithm recognizes that the area of a O1 and O3 rectangle would be much smaller than the area of a O3 and O2 rectangle.

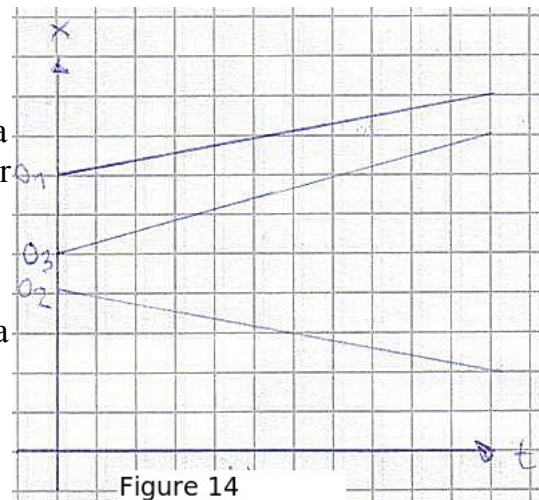


Figure 14

5.2 Minimum Bounding Rectangle

The minimum bounding rectangle of the objects O1 and O3 increasing over time. This allows to accomplish an update of the limits. The update operation manages to minimize the size of the minimum bounding rectangle. Assuming that the three objects will move constantly, an update of the minimum bounding rectangle will be useless if an update already has been done after objects O1 and O3 have crossed each other. At figure 15 the time the dashed lines get moved at the update time. This indicates that the rectangle size has decreased.

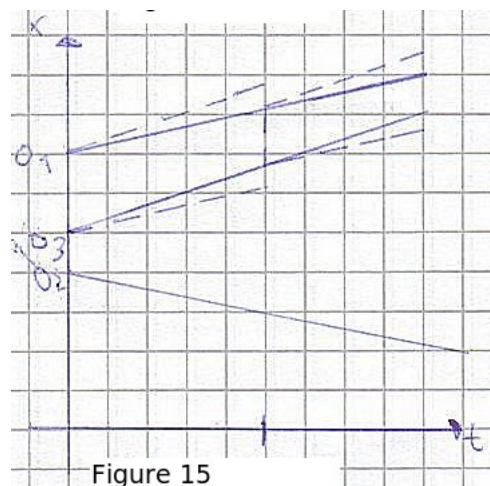


Figure 15

Figure 16 shows three different query types. The time slice query Q0 at t_1 delivers O2 after the TPR-tree has calculated its structure at t equal to t_1 .

5.4 Deletion

Figure 17

Figure 18 shows four objects A,B,C and D. The time t is equal to 0. A and B and C and D are in two minimum bounding rectangles. The figure is in two dimensions x and y . A moves

with speed 1 to the right. All other objects move as it can be seen on the picture.

At t equal to 1 the objects have moved and the minimum bounding rectangles also have moved. As is can be seen the bounding rectangles grow over time. After 2 time periods the objects have continued to move and the bounding rectangles have grown even more.

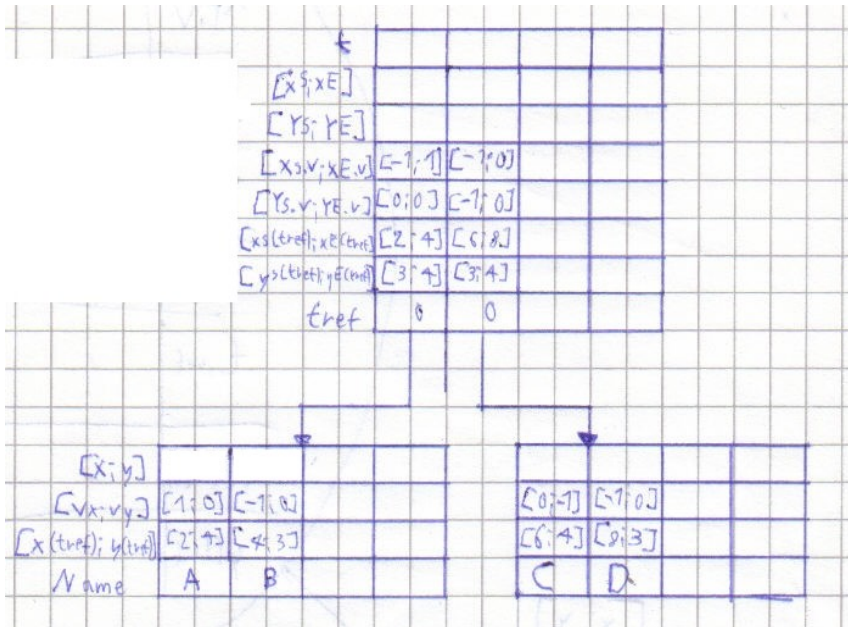


Figure 19: TPR-Tree representation

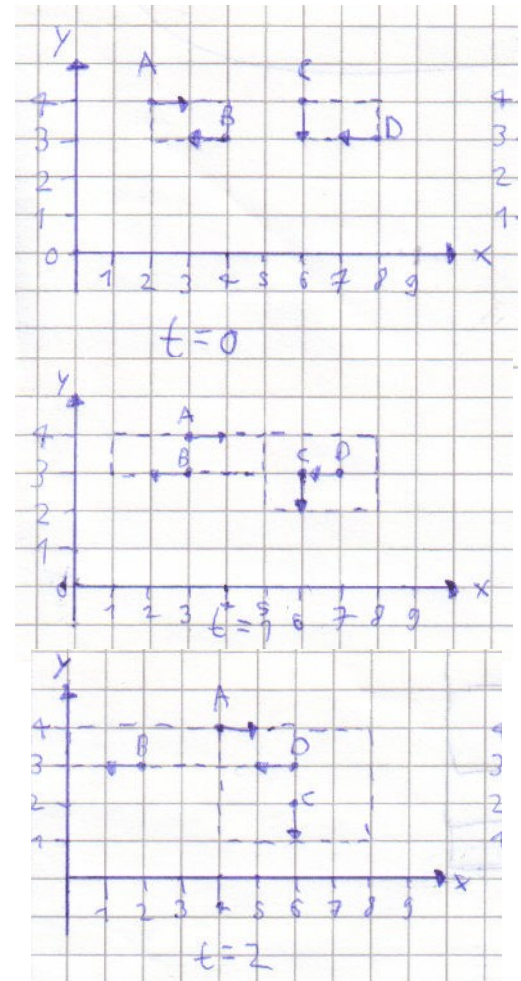


Figure 18: TPR-Tree example

Figure 19 shows the representation of the TPR-tree. The empty spaces t , $[xS;xE]$, $[yS;yE]$ and $[x,y]$ are empty because giving a specific t value automatically results in being able to calculate $[xS;xE]$, $[yS;yE]$ and $[x,y]$. This is due to the fact that the coordinates are a function of time. Actually they are not really stored in a TPR-tree, there are only in the figure in order to explain the principle. $[xs.v; xe.v]$ are the speeds of the rectangles limits in the dimension x . The left side of the first MBR is at speed -1 because the smallest speed of all objects is also -1 from object B. The right sides speed therefore is equal to 0 because this is the highest speed of all contained objects. This procedure is done for every dimension. $[xs(tref); xe(tref)]$ are the limits of the MBR in the dimension x at the reference time. The same is valid for $[ys(tref); ye(tref)]$ in the dimension y . The objects are saved into different branches with their name speed and position at reference time. Again their position can be derived from their positions at a reference time and their speed. Because of those facts the TPR-Tree in Figure 19 does not change over time, except of the fields intentionally left empty.

5.6 Querying example

Figure 20 shows an example for a moving query. Its lower y bound moves with speed -2.

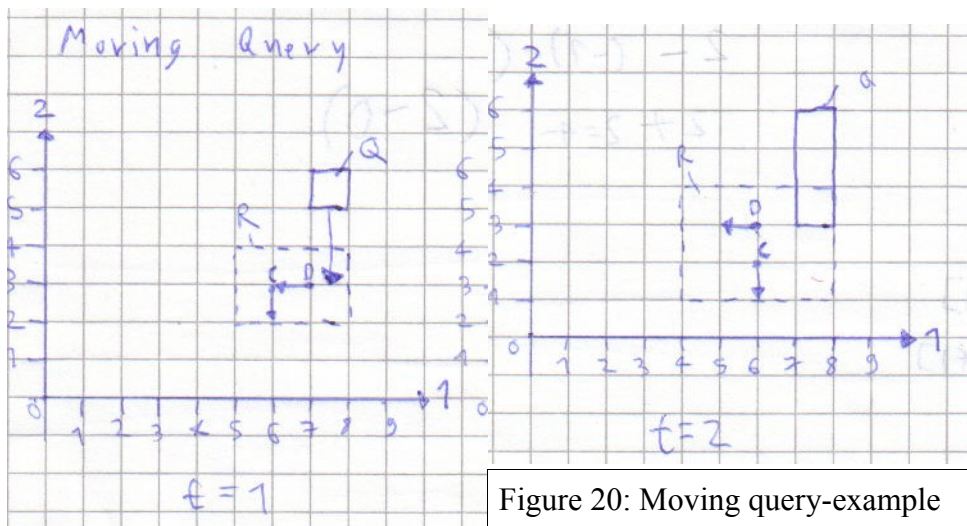


Figure 20: Moving query-example

The query and the moving rectangle are now summarized in the following query table in order to apply the query-algorithm.

	Moving Query Q		Rectangle R
$[aS1(tS);aE1(tS)]$	$[7;8]$	$[xS1(tS);xE1(tS)]$	$[5;8]$
$[aS2(tS);aE2(tS)]$	$[5;6]$	$[xS2(tS);xE2(tS)]$	$[2;4]$
$[aS1(tE);aE1(tE)]$	$[7;8]$	$[xS1(tE);xE1(tE)]$	$[4;8]$
$[aS2(tE);aE2(tE)]$	$[3;6]$	$[xS2(tE);xE2(tE)]$	$[1;4]$
$[wS1;wE1]$	$[0;0]$	$[vS1,vE1]$	$[-1;0]$
$[wS2;wE2]$	$[-2;0]$	$[vS2,vE2]$	$[-1;0]$
$[tS;tE]$	$[1;2]$		

The query algorithm now is applied for the dimension 1. Since the Boolean expression $(7 > 8 \text{ and } 7 > 8 \text{ or } 8 < 5 \text{ and } 8 < 4)$ evaluates false we have to calculate $tS1$ and $tE1$. For $tS1$ we get tS because $(7 > 8)$ and $(8 < 5)$ are both wrong. For $tE1$ we get tE because $(7 > 8)$ and $(8 < 4)$ are both wrong. So the result for $I1$ is $[1;2]$.

Now we apply the algorithm for the dimension 2. We again have to calculate $tS2$ and $tE2$ because the Boolean expression $(5 > 4 \text{ and } 3 > 4 \text{ or } 6 < 2 \text{ and } 6 < 1)$ evaluates as wrong. Because $5 > 4$ is true $tS2$ is equal to $1 + (4 - 5) / (-2 - 0) = 3/2$. $tE2$ is equal to 2 because $(3 > 4)$ and $(6 < 1)$ are both wrong and we get $I2 [3/2;2]$.

Taking the intersection of the intervals $I1$ and $I2$ we get $I = [3/2;2]$. So an object can only be found in this interval by the query. Assuming that we are only interested in integer time values we only have to check a time slice query at $t = 2$. The time slice query then returns no object for t equal 2, as can be seen in figure 20.

References.

N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and Robust Access Method for Points and Rectangles. In Proc. Of the ACM SIGMOD Conf., pp. 322-332 (1990)

J.M. Hellerstein, J.F. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In Proc. Of the VLDB Conf., pp. 562-573 (1995).

J. Moreira, C. Ribeiro, and J. Saglio. Representation and Manipulation of Moving Points: An Extended Data Model for Location Estimation. Cartography and Geographical Information Systems, (2001).

D. Pfoser and C.S. Jensen. Capturing the Uncertainty of Moving Point-Object Representations. In Proc. Of the SSDBM Conf., pp. 111-132 (1999).

O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In Proc. Of the SSDBM Conf., pp. 111-122 (1998).

O. Wolfson, A.P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that track Mobile Units. Distributed and Parallel Databases 7(3): 257-387 (1999).

S. Saltenis, C.S. Jensen, S.T. Leutenegger, M.A. Lopez. Indexing the Positions of continuously Moving Objects. ACM SIGMOD Record 331-342 (2000).