Indexing the Positions of Continuously Moving Objects

Simonas Šaltenis   Christian S. Jensen

Aalborg University, Denmark

Scott T. Leutenegger   Mario A. Lopez

Denver University, USA

SIGMOD 2000

presented by

Simonas Šaltenis

Why Moving Objects?

- Position-aware, online, moving objects are enabled by the following trends.
  - Miniaturization of electronics
  - Advances in positioning systems (e.g., GPS, assisted GPS, ...)
  - Advances in wireless communications

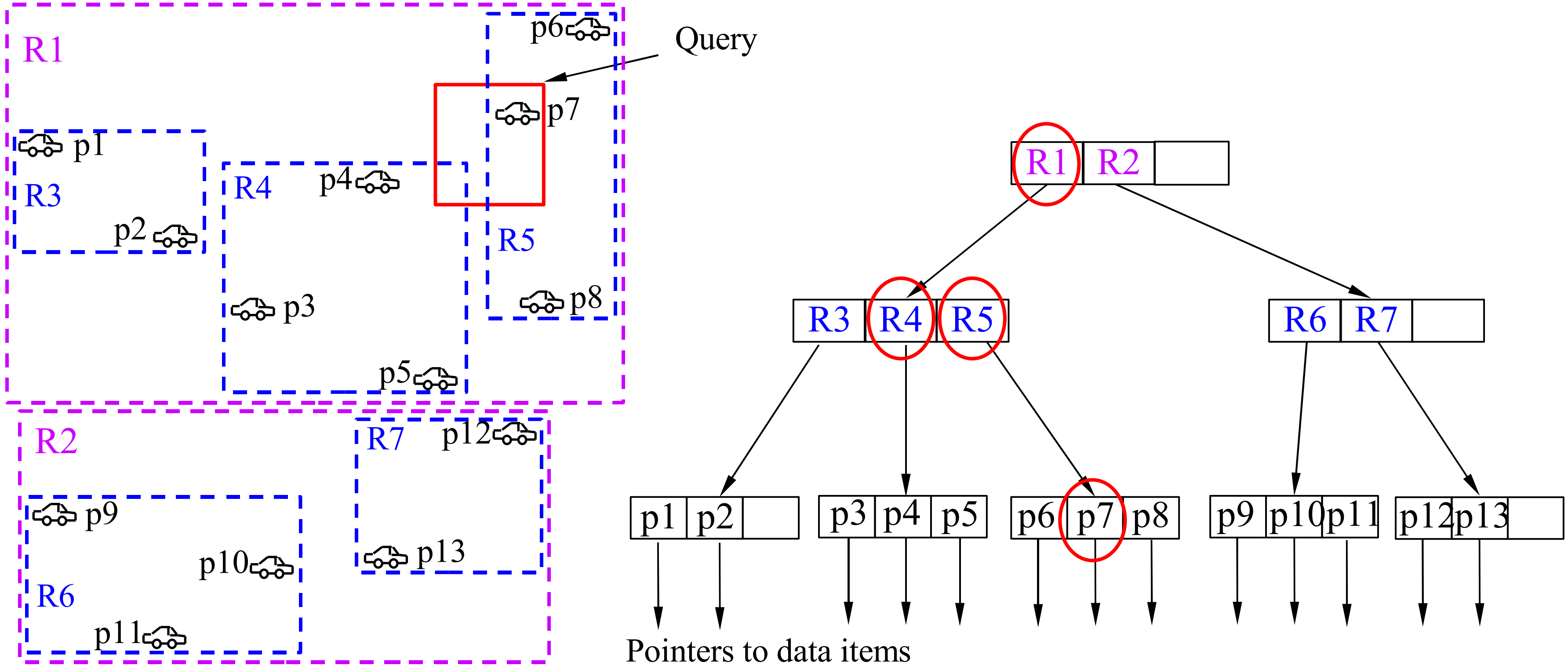
- Examples of position-aware online moving objects
  - GPS-enabled mobile-phones, as well as diverse types of personal digital assistants (online "cameras," "wrist watches," etc.).
    - The coming years will witness very large quantities of these.
  - Vehicles, including cars, public transportation, recreational vehicles, sea vessels, airplanes, etc.
- Sensor-networks also generate MO data
  - Monitoring of any kind-of continuous variables, e.g., temperature, pressure

Outline

- Motivation
- Background: R-tree
- Problem definition
  - Data and queries
- Structure and algorithms of the TPR-tree
- Insertion example
- Summary

Spatial Indexing With the R-Tree

- Example



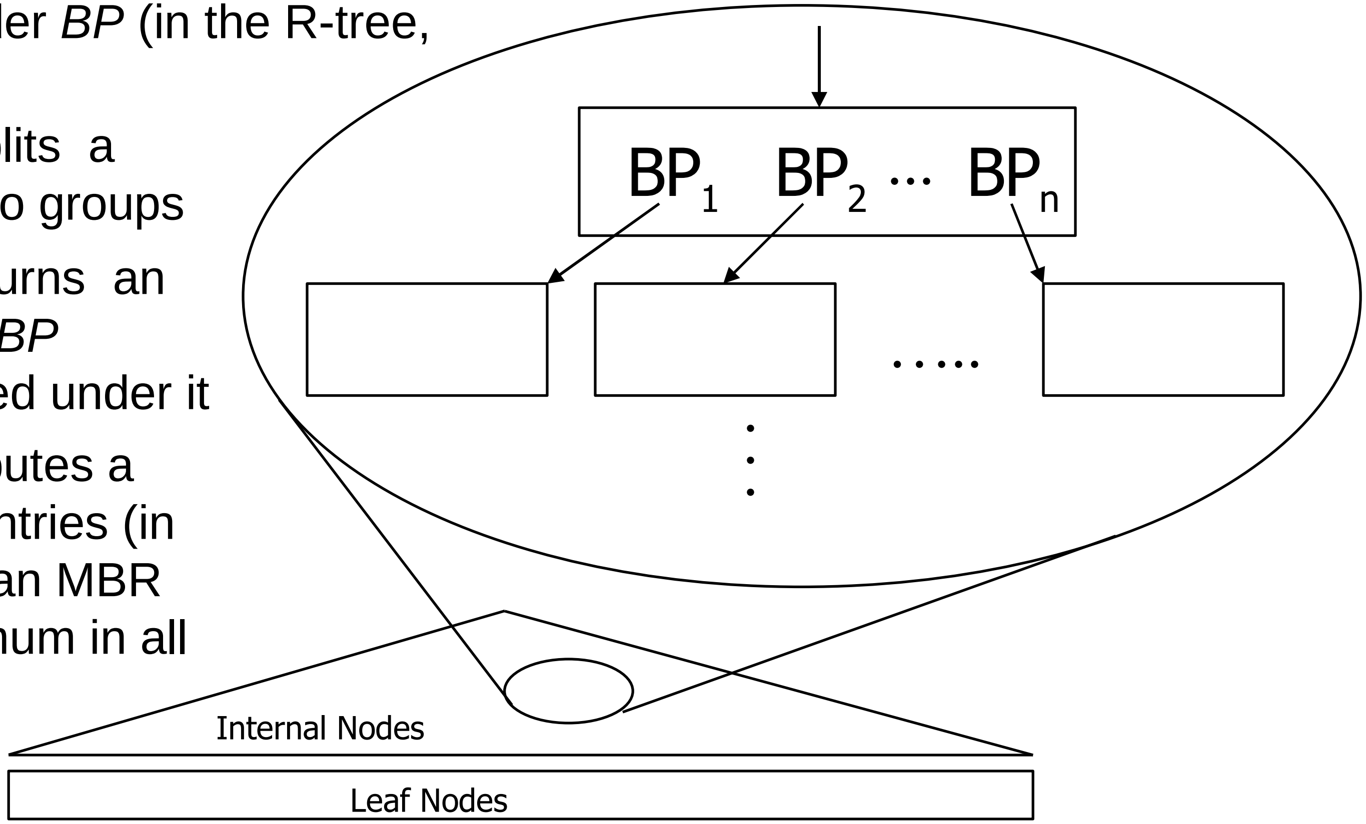
Grow-Post trees

Grow-Post trees: generalized R-tree-type indexes

Bounding predicate (*BP*) = *something that describes entries in a subtree*

Building blocks of algorithms:

- **Consistent**(*BP*, *Q*) – returns *true* if results of query *Q* can be under *BP* (in the R-tree, MBR intersects *Q*)
- **PickSplit**(*node*) – splits a page of entries into two groups
- **Penalty**(*BP, E*) – returns an estimate how "worse" *BP* becomes if *E* is inserted under it
- **Union**(*node*) – computes a BP of a collection of entries (in the R-tree, computes an MBR – minimum and maximum in all dimensions)

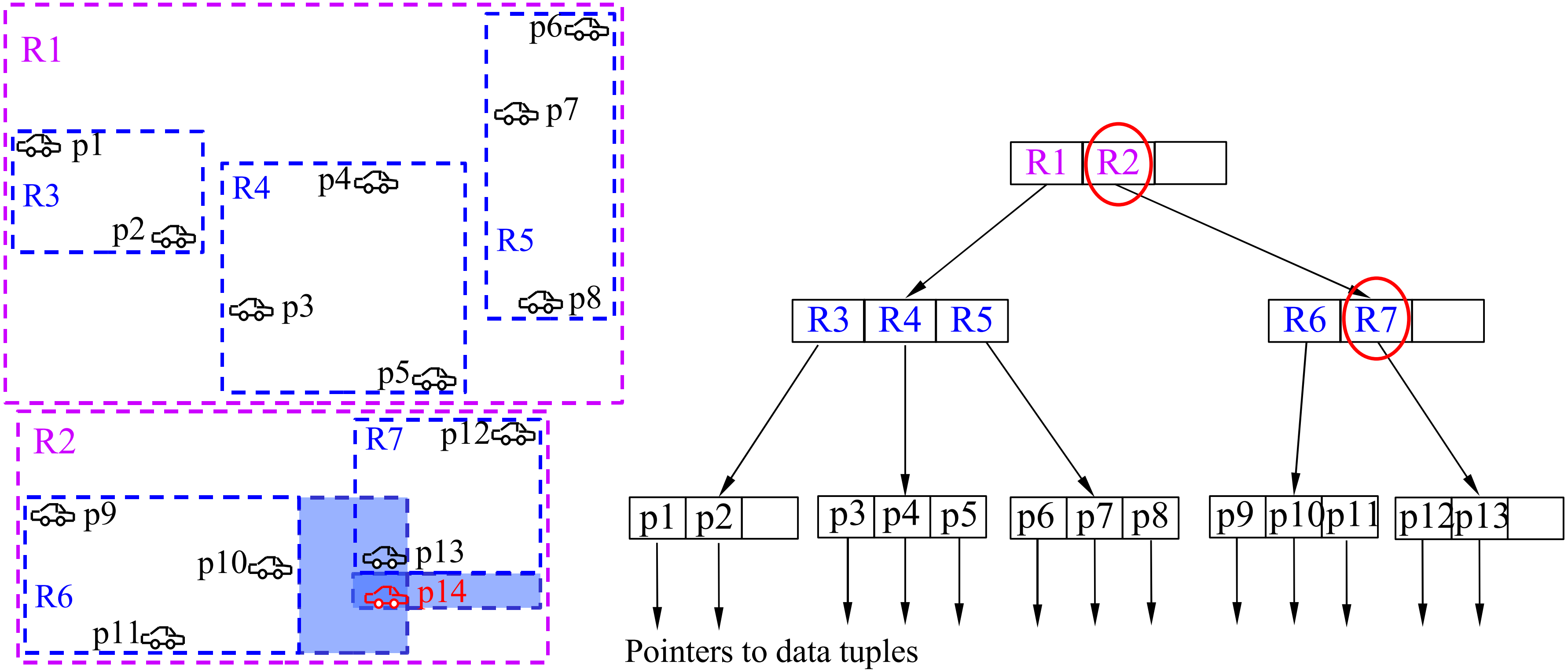


Insertion

- *Insert*(*E*)
  - *leaf* = *ChoosePath*(*E*, *root*)
  - Insert *E* into *leaf*
  - *PropogateUp* (*leaf*)
- *ChoosePath*(*E*, *node*)
  - **If** *node* is leaf, **return** *node*.
  - From all entries in *node,* choose entry *<MBR, ptr>* with the smallest **Penalty (***MBR, E***).**
  - *ChoosePath*(*E*, *ReadNode*(*ptr*)).
- *PropogateUp*(*node*)
  - **If** *node* is overfull, call **PickSplit(***node***)** to produce *n1* and *n2*, replace *node's* old entry in its parent by *e1* = **Union(***n1***),** *e2* = **Union(***n2***)**, call *PropogateUp*(*node's* parent)
  - **Else if** *e* = **Union(***node***)** is different from *node's* old entry in its parent, replace the old entry with *e,* call *PropogateUp*(*node*'s parent).
- Create a new root with two entries whenever a root is split.

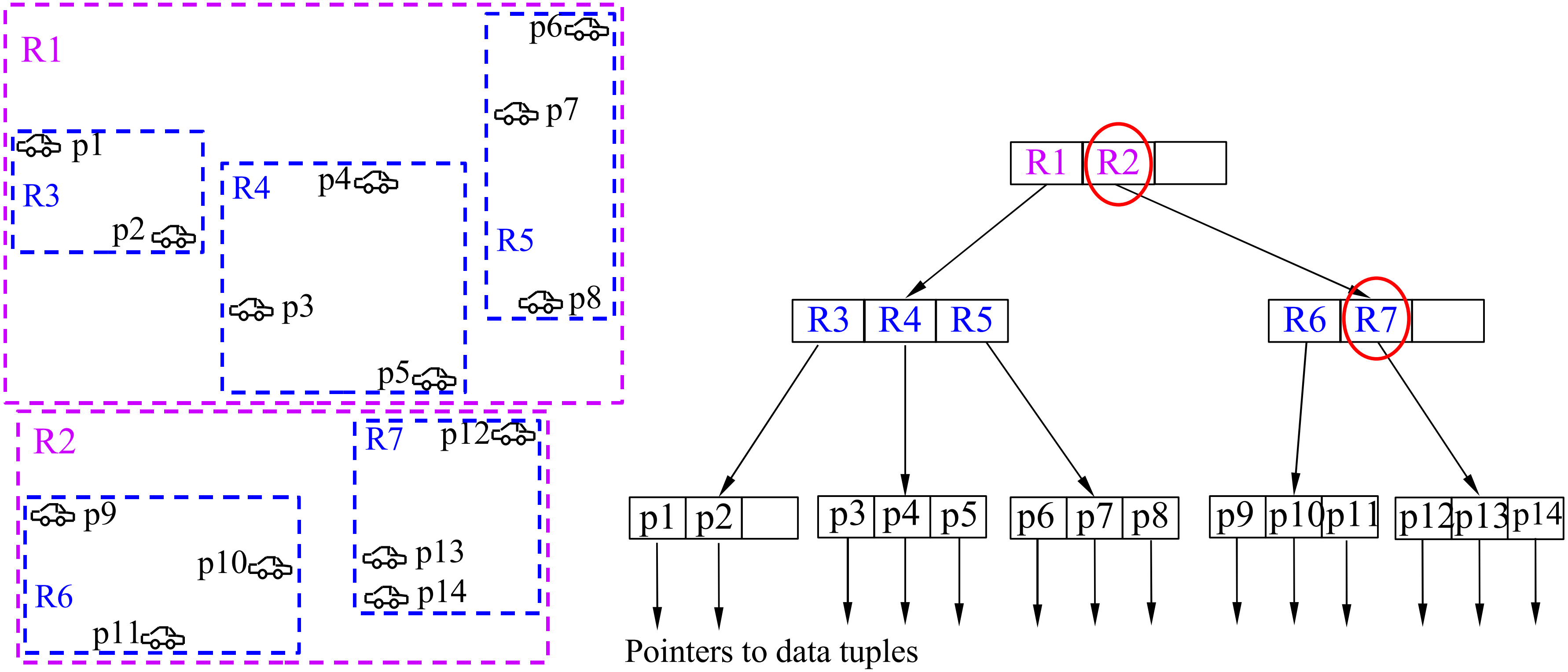
Heuristics for **Penalty**

- Heuristics of *least area enlargement* and *smallest area* are used in the R-tree's **Penalty**.



Heuristics for **Penalty**

- Heuristics of *least area enlargement* and *smallest area* are used in the R-tree's **Penalty**.



Comments on R-Trees

- Works well for 2 - 4 D datasets. Several variants (notably, R^+ and R*-trees) have been proposed; widely used
- Supports a wide variety of queries
  - Point / range queries
  - Spatial join queries [Brinkhoff et al., 1993]
  - Direction, topological, distance queries [Papadias et al., 1995]
  - k- Nearest neighbor queries [Roussopoulos et al., 1995]

Outline

- Motivation
- Background: R-tree
- Problem definition
  - Data and queries
- Structure and algorithms of the TPR-tree
- Insertion example
- Summary

Problem Statement

We address the problem of indexing the ever-changing current and predicted future positions of point objects moving in one, two, and three-dimensional space.

- Indexing challenges specific to MOs
  - continuous change of positions – extrapolation between the last update and the current time must be supported
  - hyper-dynamic workloads – high-rates of updates

Modeling Continuous Movement

- In conventional databases, data is assumed constant unless explicitly modified.
- With continuous movement, this is problematic.
  - Too frequent updates
  - Outdated, inacurate data

Modeling Continuous Movement

- In conventional databases, data is assumed constant unless explicitly modified.
- With continuous movement, this is problematic.
  - Too frequent updates
  - Outdated, inacurate data
- Instead of storing position values, we store positions as functions of time, yielding *time-parameterized* positions.
  - We use linear functions to capture the present and future positions.

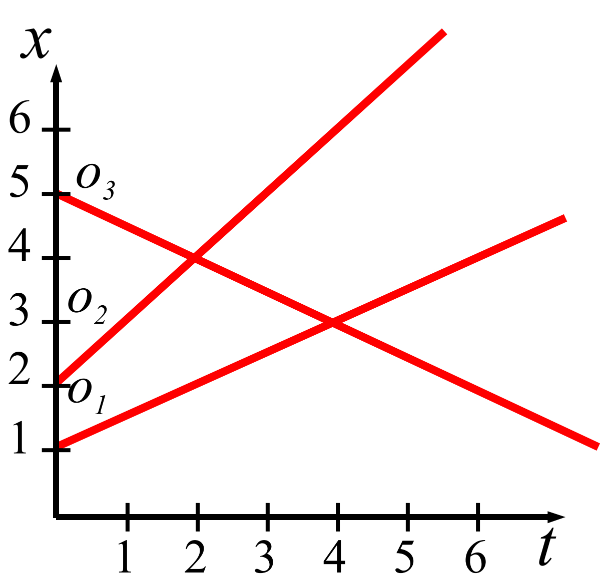
$$\bar{x}(t) = \bar{x}(t_0) + \bar{v}(t - t_0), \quad \text{where } t \geq \mathit{now}$$

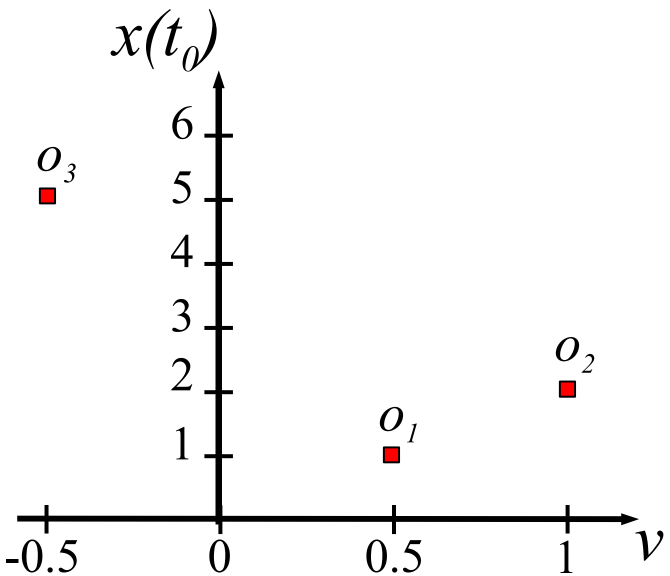
  - Updates are less frequent
  - Tentative future queries are supported
  - For example, given t_0, the current and anticiapted, future position of a two-dimensional point can be described by four parameters.

$$x(t_0), y(t_0), v_x, v_y$$

Modeling Continuous Movement

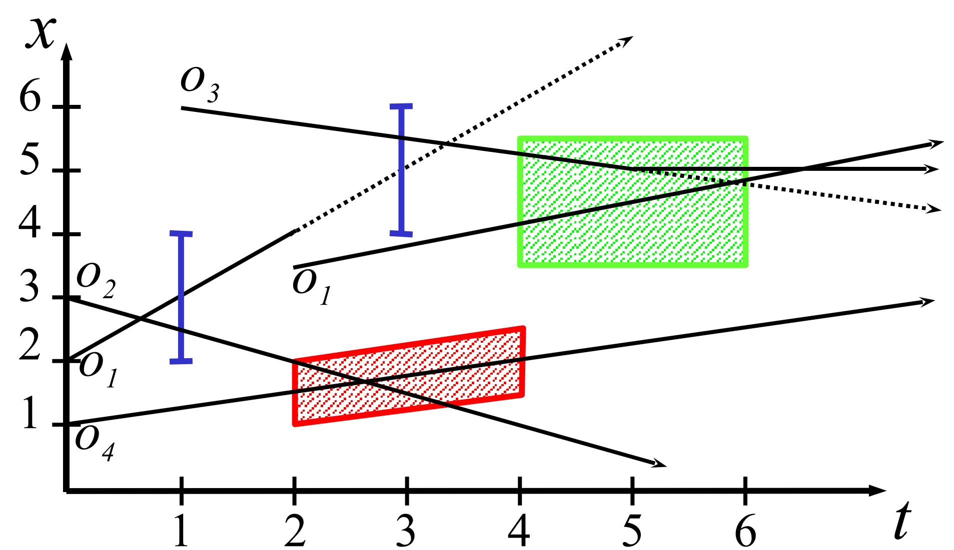
- Three ways to think about continuously moving points in *d*-dimensional space:
  - Lines in (*d*+1)-dimensional space
    - *d* spatial dimensions and 1 time dimension
  - Points in 2*d*-dimensional space
    - *d* spatial and *d* velocity dimensions (function parameters: $\bar{x}(t_0), \bar{v}$)
  - Time-parameterized points in *d*-dimensional space





Queries

- **Type 1**: objects that intersect a given rectangle at t
- **Type 2**: objects that intersect a given rectangle sometime from t_1 to t_2
- **Type 3**: objects that intersect a given moving rectangle sometime between t_1 and t_2



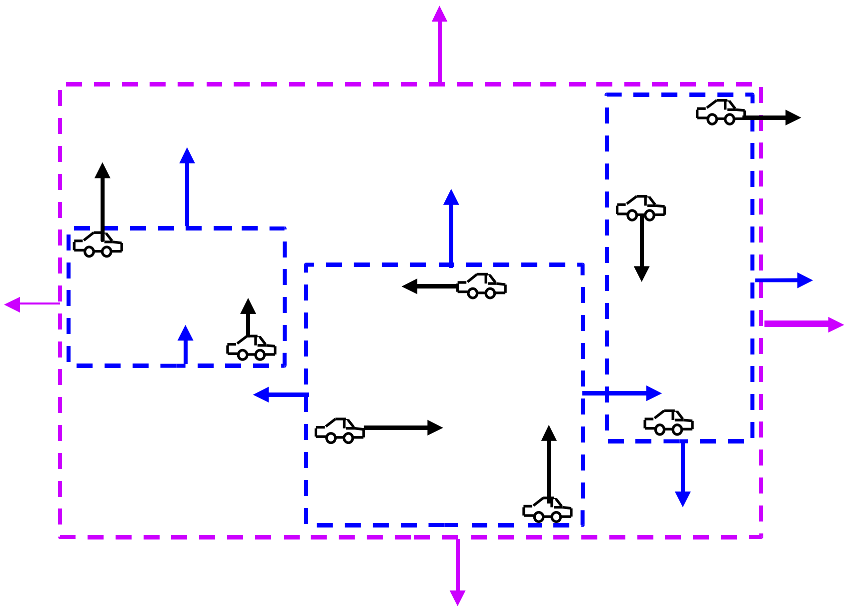
- We can expect, that most queries will be consentrated in the sliding window [CT, CT+*W*], i.e. $CT \le t, t_1, t_2 \le CT + W$

Outline

- Motivation
- Background: R-tree
- Problem definition
  - Data and queries
- Structure and algorithms of the TPR-tree
- Insertion example
- Summary

Time-Parameterized Rectangles

- The TPR-tree is based on the R-tree.
- Moving points are bounded with *time-parameterized* rectangles.
  - Are bounding from *now* on.
  - The R-tree allows overlap.
- The tree employs *conservative bounding rectangles.*



$$\text{Union}(node):$$

$$x_i^{\min}(t_c) = \min\{o.x_i(t_c) \mid o \in node\}$$

$$x_i^{\max}(t_c) = \max\{o.x_i(t_c) \mid o \in node\}$$

$$v_i^{\min} = \min\{o.v_i \mid o \in node\}$$

$$v_i^{\max} = \max\{o.v_i \mid o \in node\}$$

Entry Structure, Querying

- Do we need to store t_c in each entry?
  - No – we use one common reference time t_r (the same for data points):

$$x_i^{\min} = x_i^{\min}(t_r) = x_i^{\min}(t_c) + v_i^{\min}(t_r - t_c)$$

$$x_i^{\max} = x_i^{\max}(t_r) = x_i^{\max}(t_c) + v_i^{\max}(t_r - t_c)$$

- Entry structure: <*TPBR, ptr*>
- *TPBR* = *MBR*,*VBR* = $(x_1^{\min}, x_1^{\max}, x_2^{\min}, x_2^{\max}), (v_1^{\min}, v_1^{\max}, v_2^{\min}, v_2^{\max})$

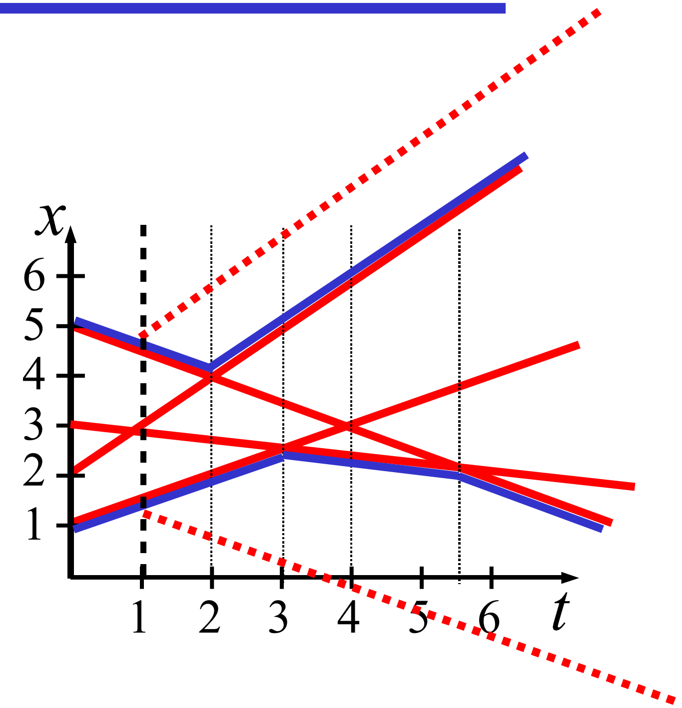
- At any $t > CT$ we can get a valid R-tree: TPR-tree*(t)* = R-tree

$$x_i^{\min}(t) = x_i^{\min}(t_r) + v_i^{\min}(t - t_r)$$

$$x_i^{\max}(t) = x_i^{\max}(t_r) + v_i^{\max}(t - t_r)$$

Tightening

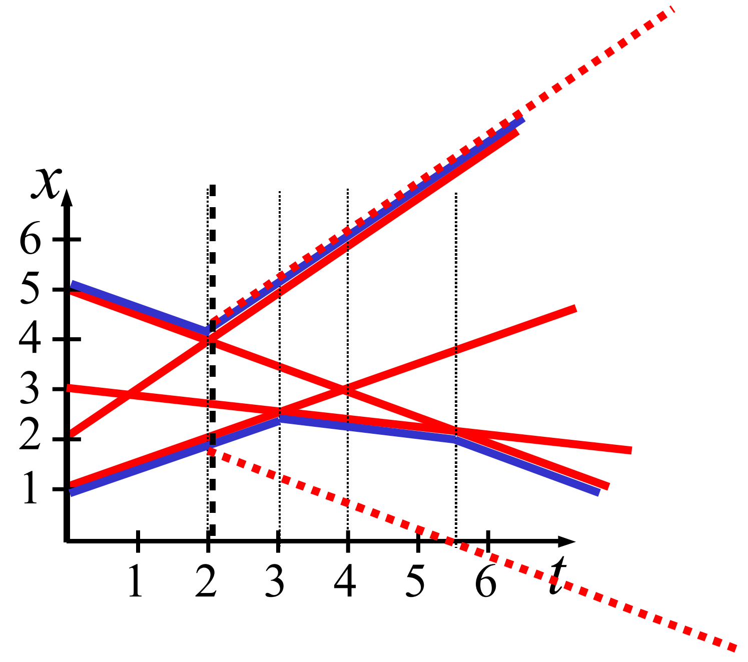
- Ideally, bounding rectangles should be always minimal.
  - Excessive storage cost



- TPBRs are ”*tightened*” (i.e., recomputed) whenever a bounded node is modified

Tightening

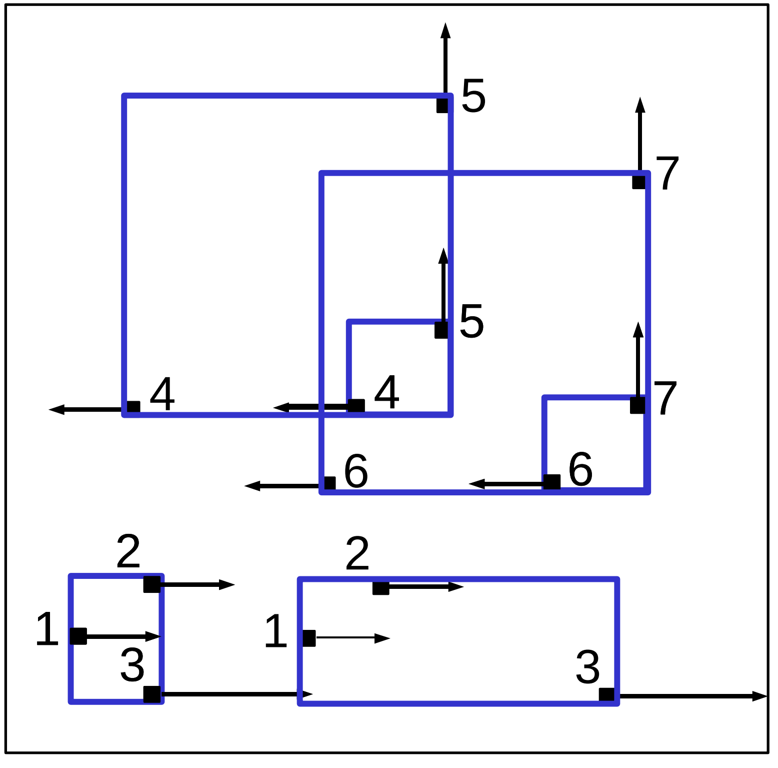
- Ideally, bounding rectangles should be always minimal.
  - Excessive storage cost

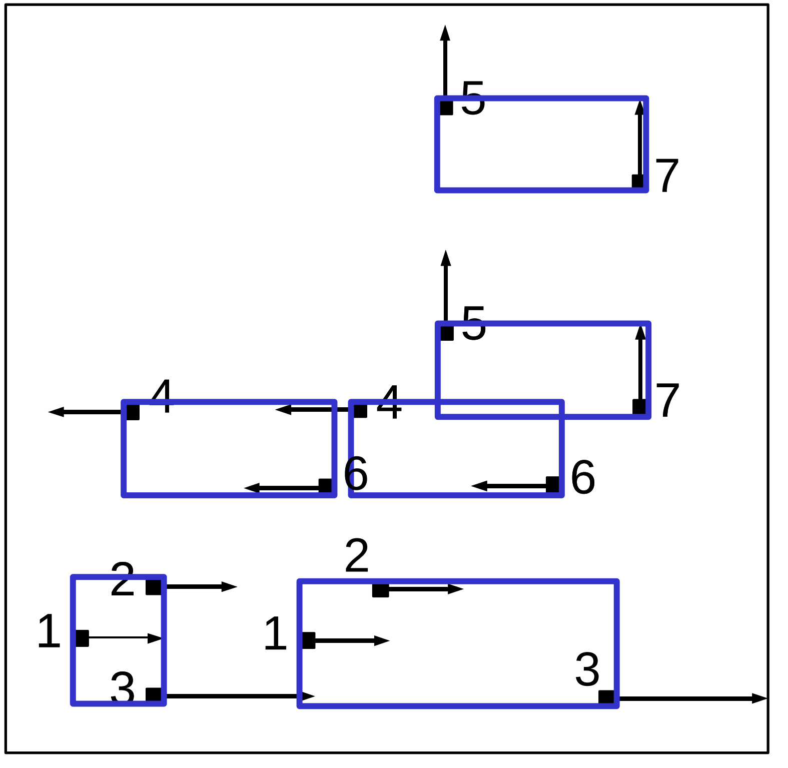


- TPBRs are ”*tightened*” (i.e., recomputed) whenever a bounded node is modified

Insertion: Grouping Points

- How to group moving points (**Penalty** and **PickSplit**)?
  - The R-tree's algorithms minimize characteristics of MBRs such as area, overlap, and margin.
  - How does that work for moving points?





Insertion in the TPR-Tree

- The bounding rectangle characteristics (area, overlap, and margin) are functions of time.
- The goal is to minimize these for all time points from *now* to *now+H.*
  - Minimizing the characteristics for time *now* + *H/2* does not work (e.g., the area of a conservative bounding rectangle is not linear).
- We use the regular R*-tree algorithms, but all bounding rectangle characteristics are replaced by their *integrals*.

$$\int_{now}^{now+H} A(t)\,dt,$$ where *A*(*t*) is, e.g., the area of an MBR

What *H* to use?

- Intuitivly: we want *H* to be equal to the time during which queries will see the node that we consider modifying:
  - *H* depends on the update rate, and on how far queries may reach into the future (*W*)
  - Experiments show that *H* = *UI* + *W* consistently gives good query performance (*UI* – average update interval)
  - The system can track *UI* automatically (*How*?)
  - *W* is usually smaller than *UI*
    - can be tracked automatically too

Outline

- Motivation
- Background: R-tree
- Problem definition
  - Data and queries
- Structure and algorithms of the TPR-tree
- Insertion example
- Summary

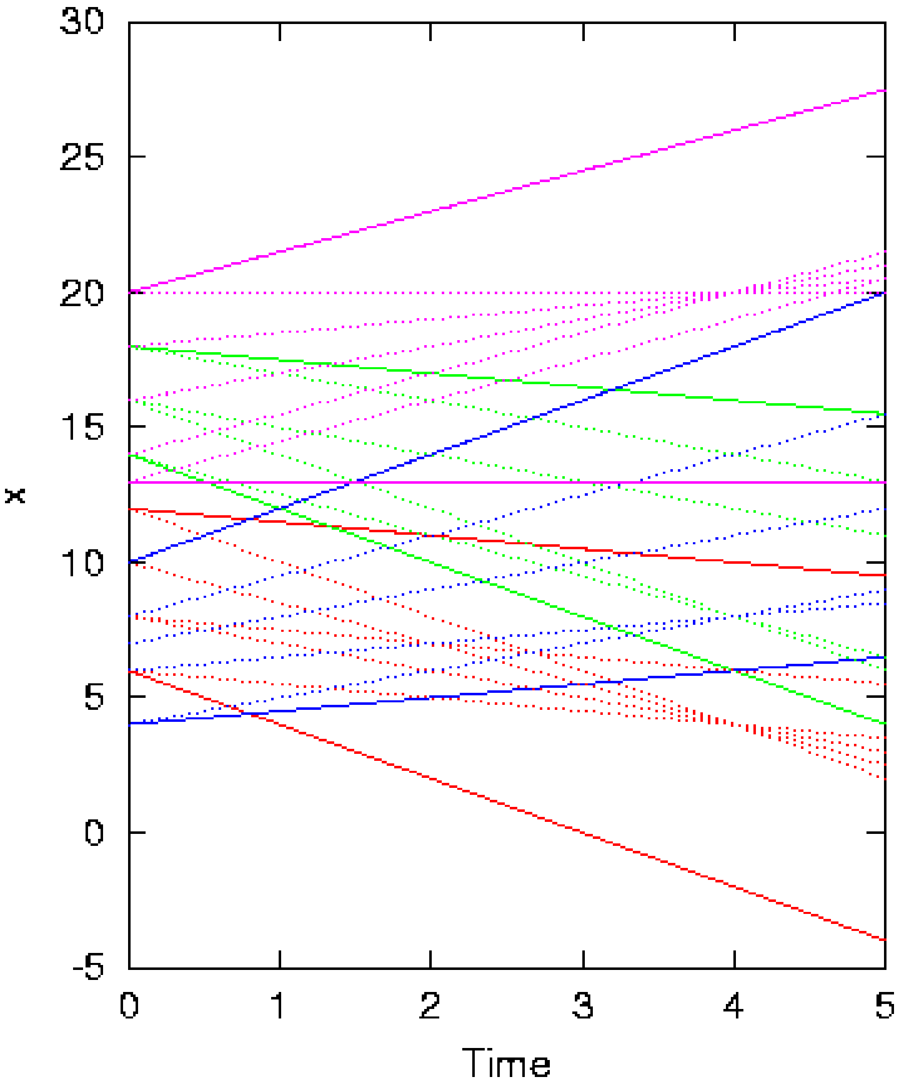
Example I

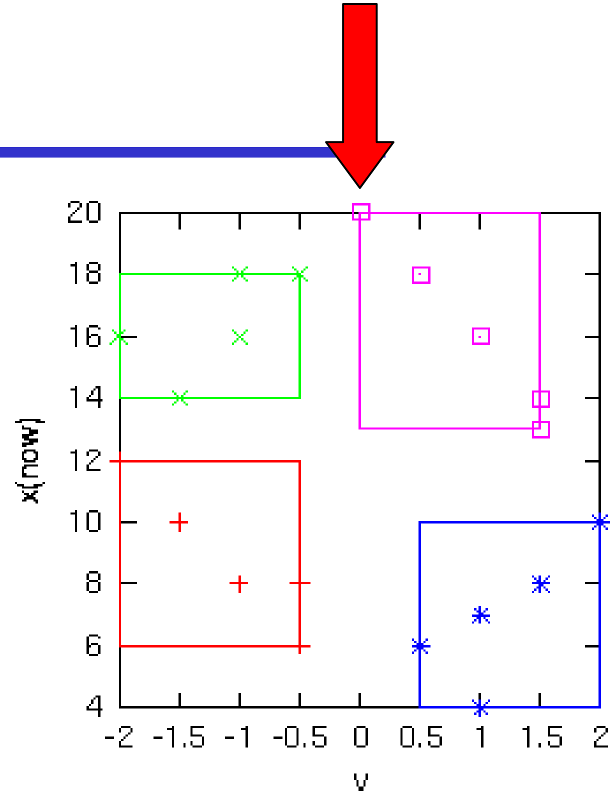
- We illustrate the working of the TPR-tree by means of an example.
  - The subsequent figures are generated automatically, by the index code used for performance experiments.

- Data
  - 20 one-dimensional points are used.

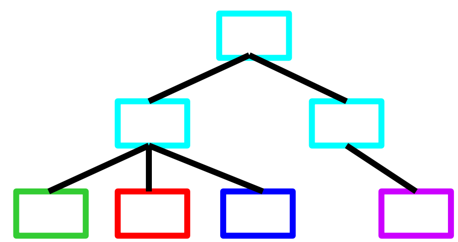
- Index Parameters
  - Page size = 64 (5 entries in leaf nodes and 3 in non-leaf nodes).
  - H = 8.

Example II



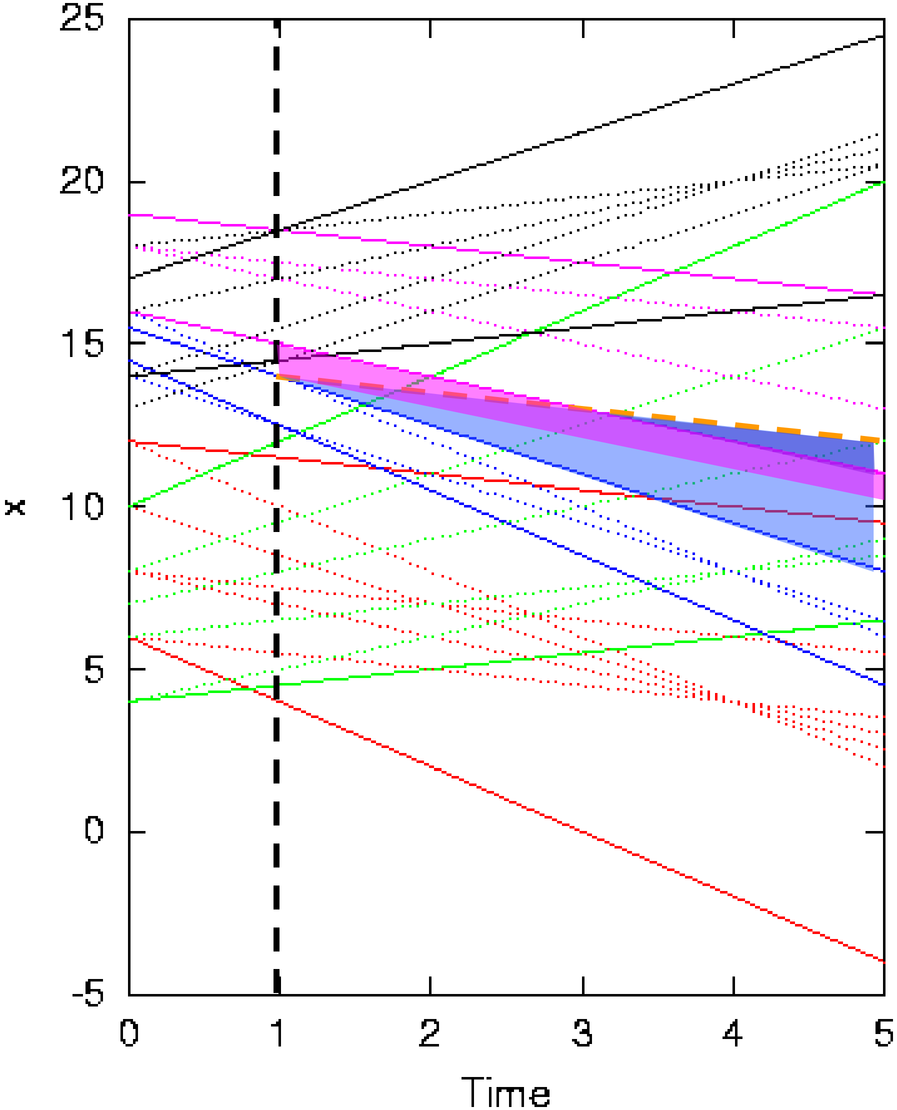


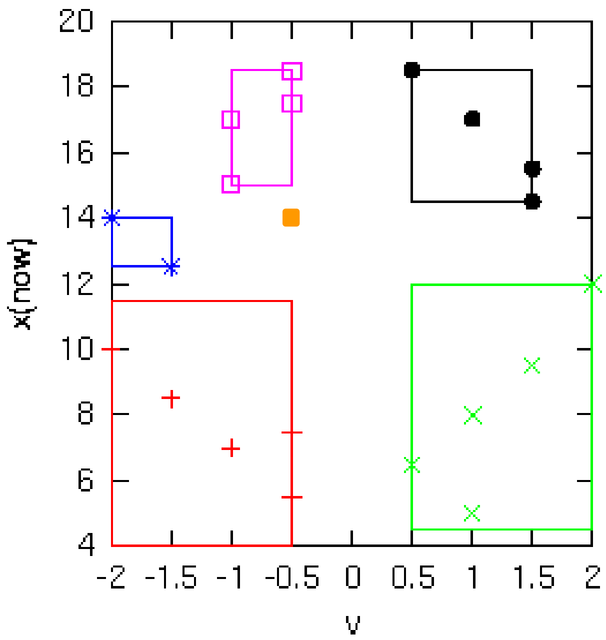
CT = 0



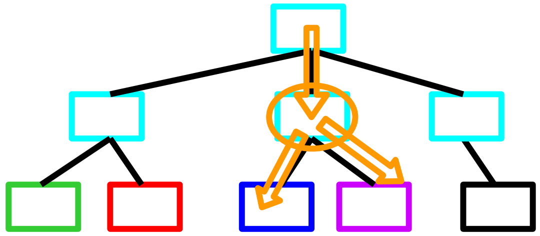
At CT=1, the point at x = 20, *v* = 0 is updated to have x = 18.5, *v* = -0.5.

Example III



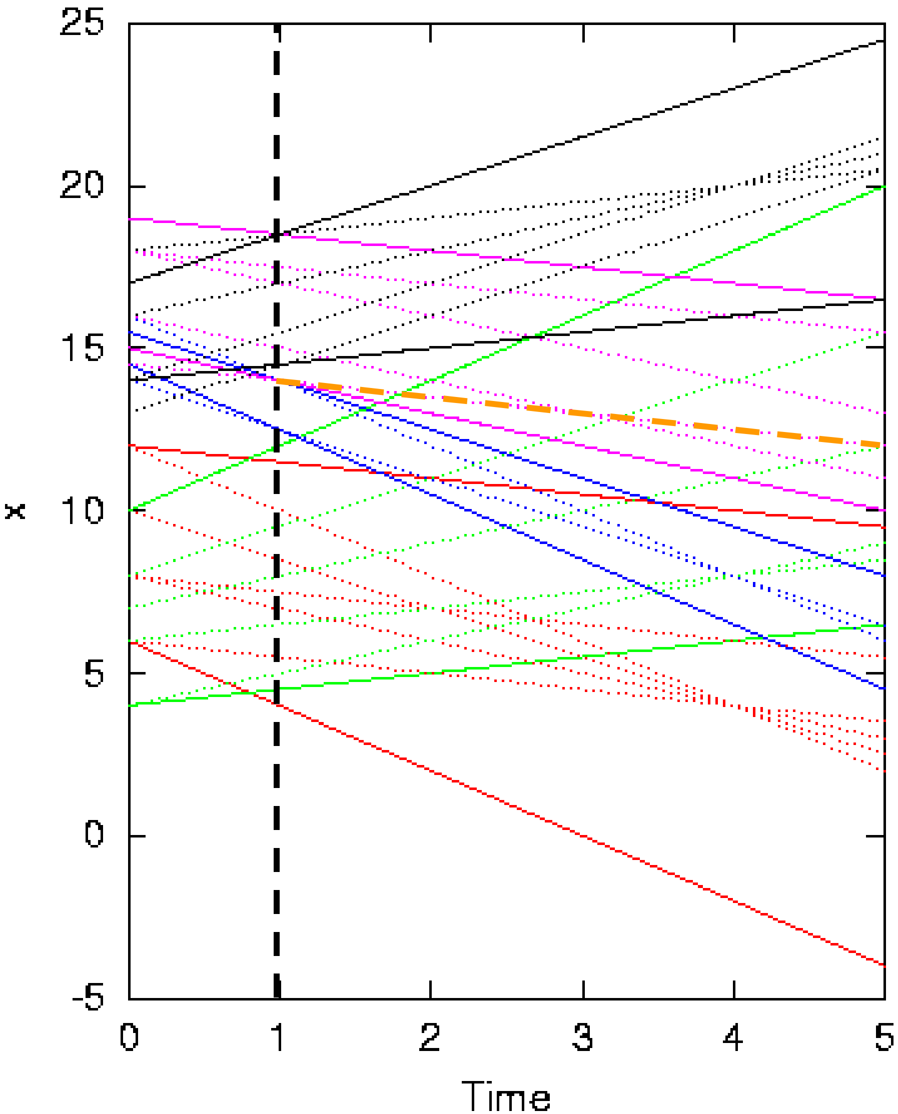


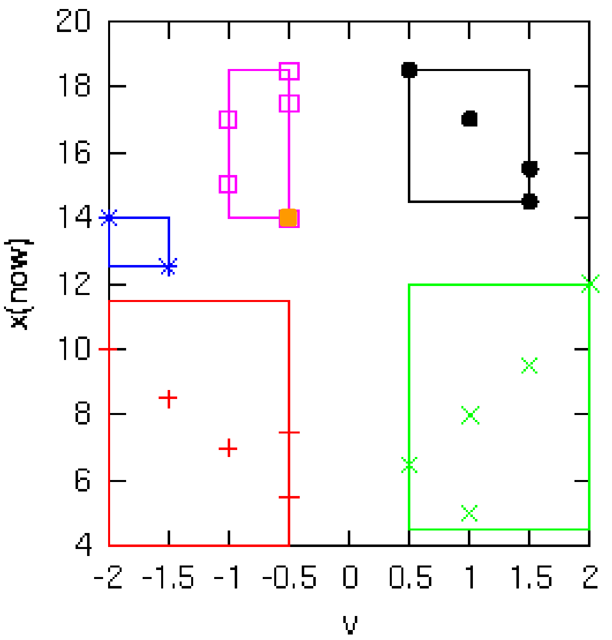
CT = 1



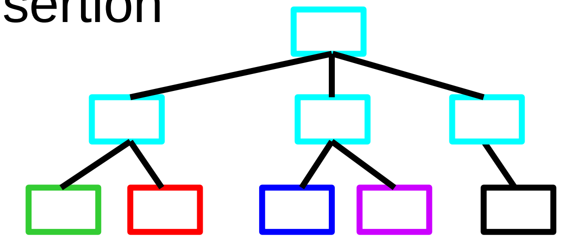
Inserting a moving point at position 14 with v = –0.5.

Example IV



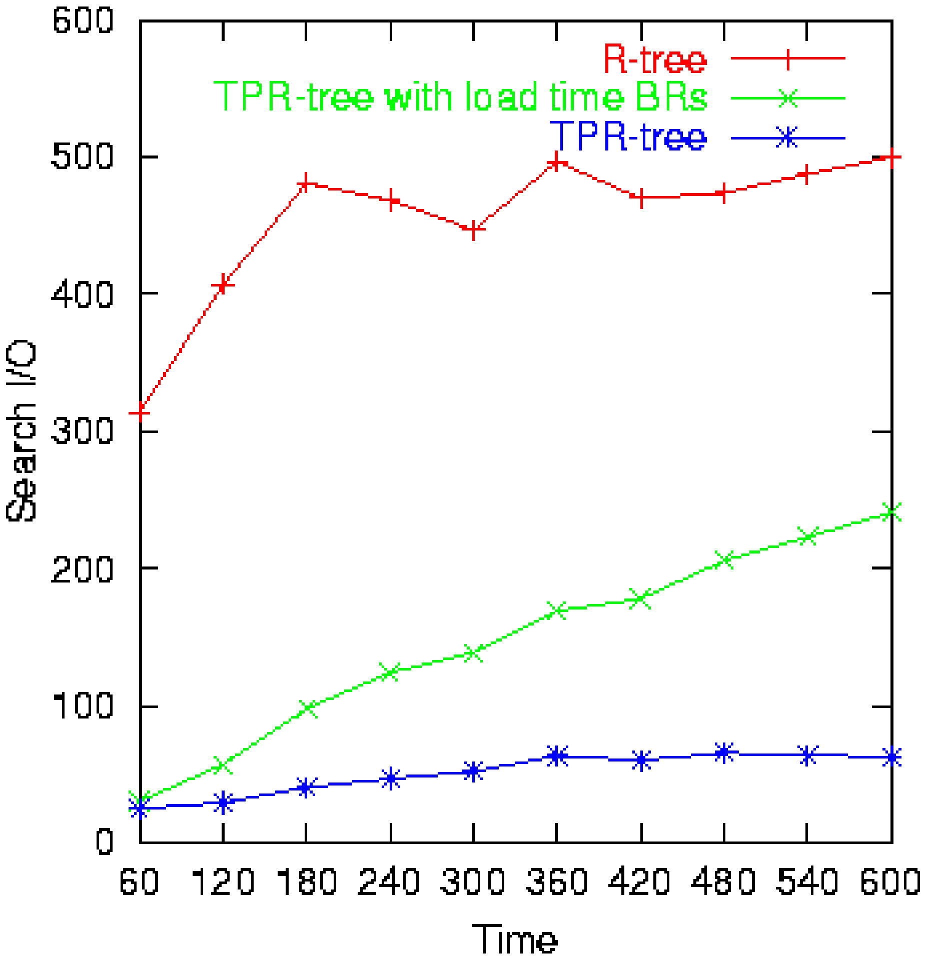


After insertion



What about Expanding BRs?

- Will the expanding TPBRs ruin the performance?
- That's what the experiments show:
  - Settings:
    - objects update only once per hour!
    - 2D data, with *W* = 40.
  - Due to the constant influx of updates, the performance of the TPR-tree does not degrade after reaching a certain level.



Summary

- The TPR-tree indexes the current and predicted future positions of moving objects.
  - The TPR-tree is based on the proven, widely used R-tree technology
  - The tree extends the R*-tree by introducing conservative, time-parameterized bounding rectangles, which are tightened regularly.
  - The tree’s algorithms use integrals of area, overlap, etc.
  - The tree can be tuned to take advantage of a specific update rate and querying window length.
  - Other types of queries that are supported by the R-tree can be supported by the TPR-tree, e.g., nearest-neighbor queries.