

Indexing the position of moving Objects



Overview

- Introduction
- Problem Setting
- R & TPR - Tree
- Querying
- Querying Example
- (Performance Tests)
- (additional Examples)

Introduction

- Difficulties of saving moving objects in a classical database
 - Two bad solutions
 - Save Positions very Frequent => overstress for the database
 - Save positions rarely => positions are often antiqued
- Both solutions clearly classify as dissatisfying
=> Represent the Object Position as function of time

Problem Setting

- An Object's Position in each dimension are defined as

$$x(t) = (x_1(t), x_2(t), \dots, x_d(t))$$

- The Object's reference position is given at a certain time t_{ref}

- The Object has velocity vectors

$$v = (v_1, v_2, \dots, v_d)$$

- The Objects position at t is

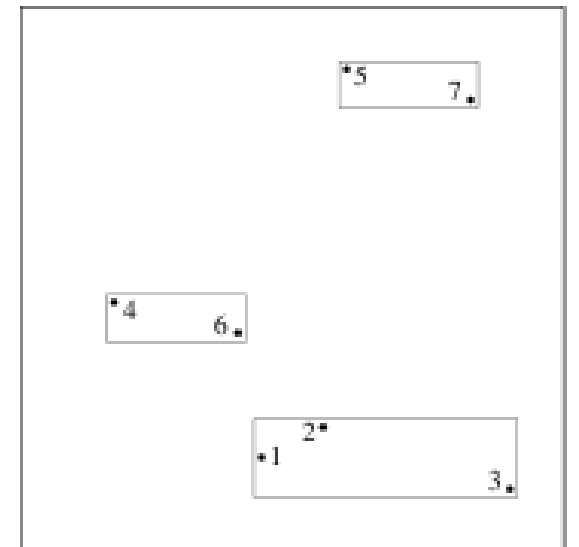
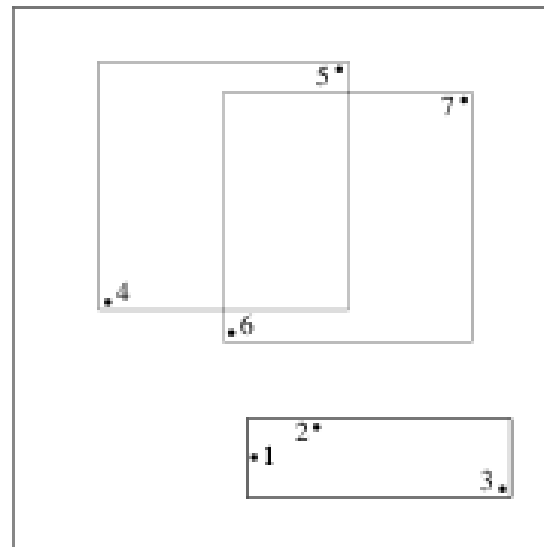
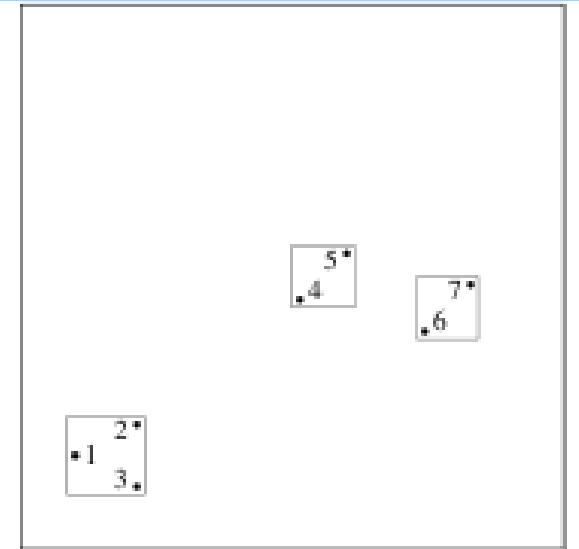
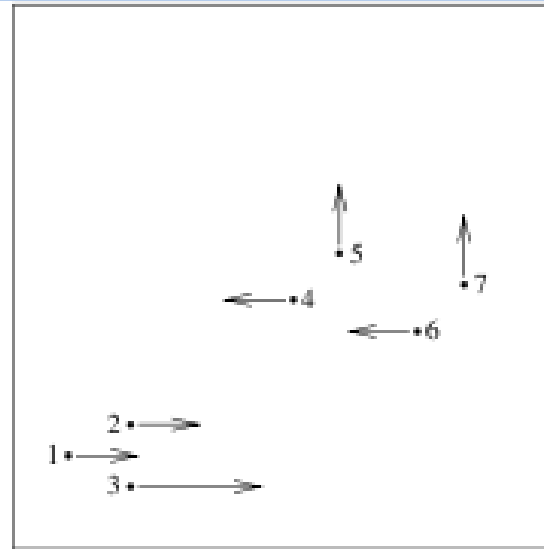
$$x(t) = x(t_{ref}) + v * (t - t_{ref})$$

Problem setting

- Idea: Save Objects movement as a linear function
- Advantage: Objects position has not to be saved at anytime. Only changements in it's movement have to be saved

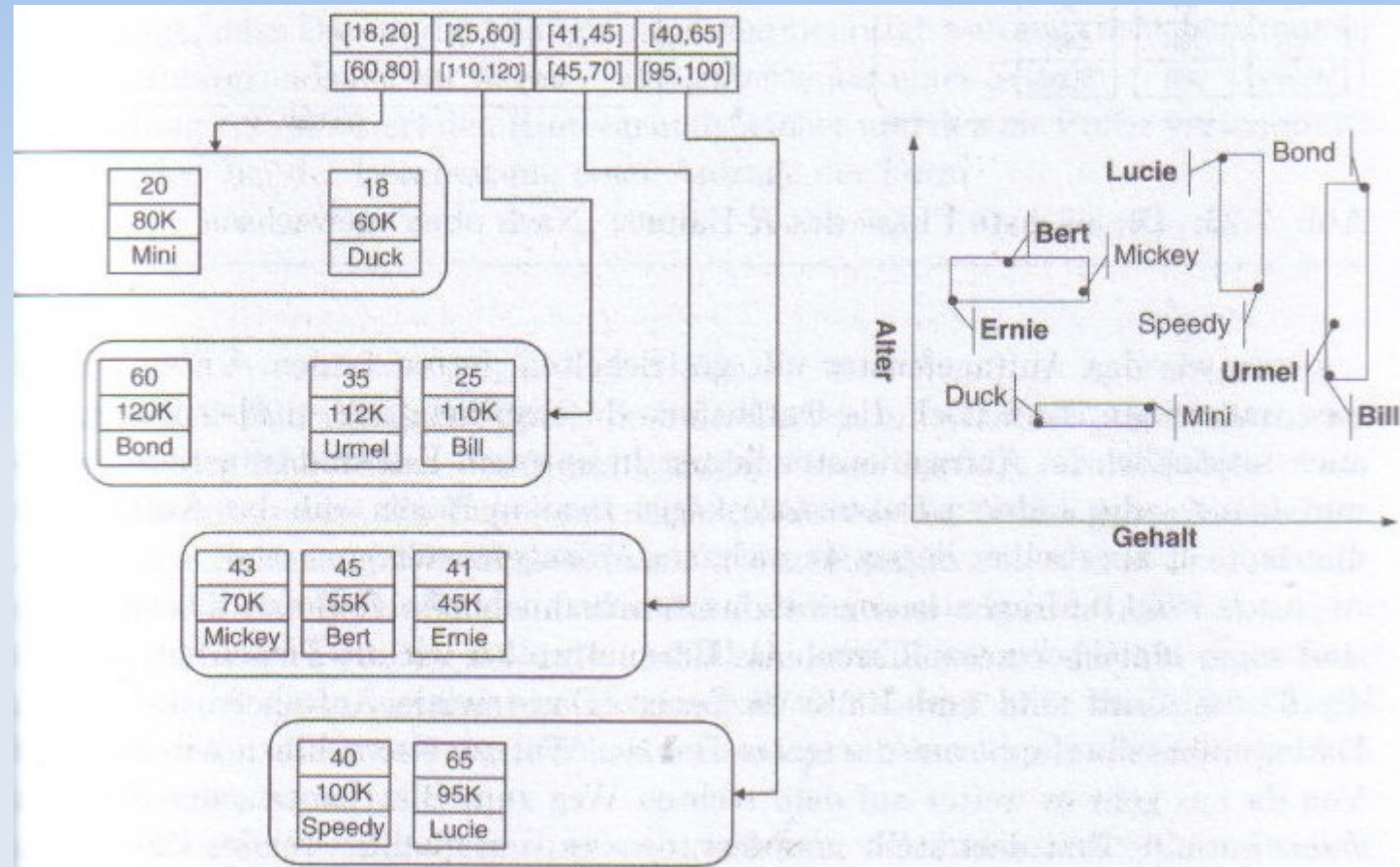
Problem setting

- Data structure to save is a Time parametrized R-Tree (TPR-Tree) based on a R-Tree
- It uses Minimum Bounding rectangles to enclose objects at any time

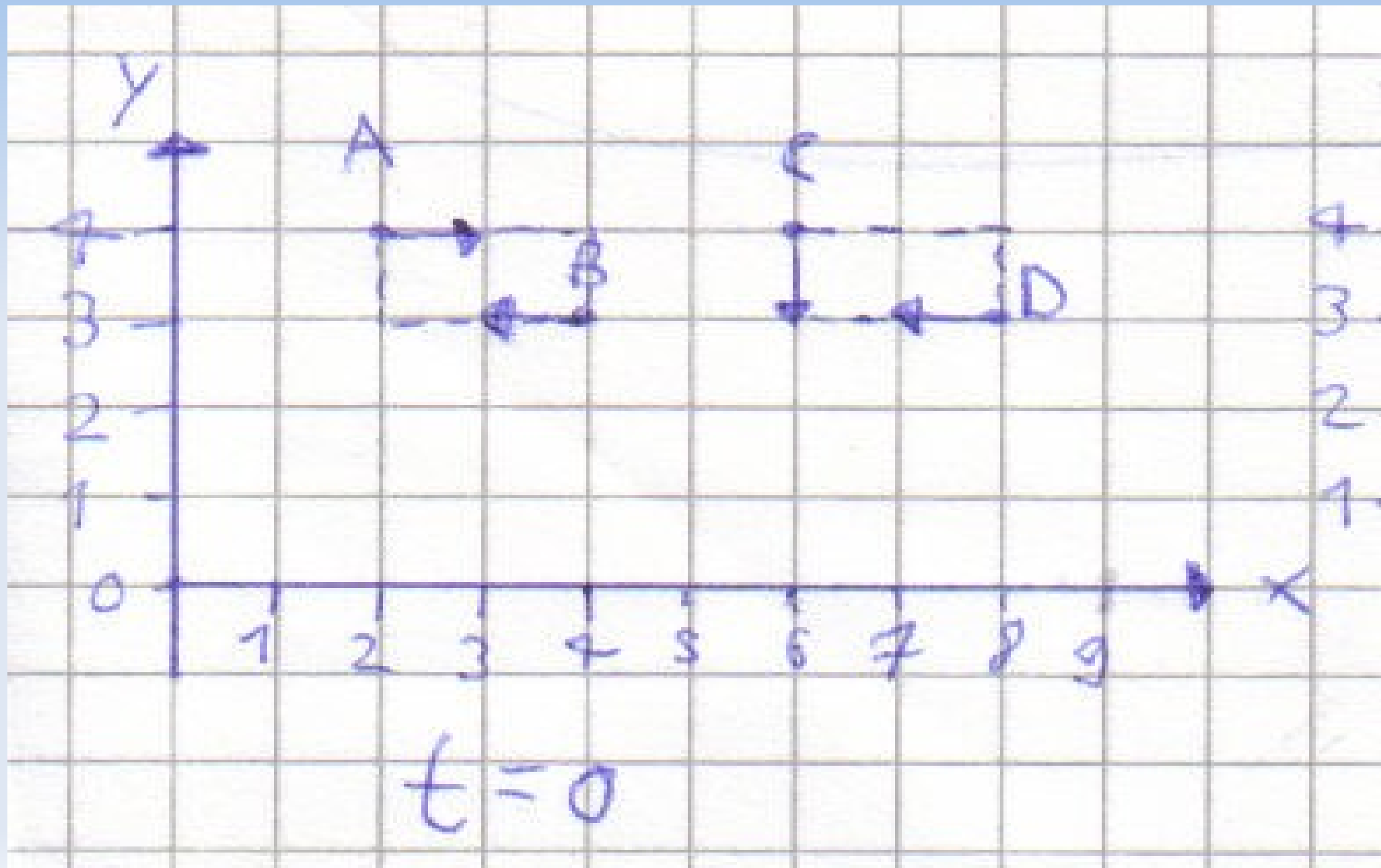


Refreshment R-Tree

- Static
- Constant Rectangles
- Efficient area Querying
- Problems with moving Objects



TPR-Tree example



Index Structure

- The minimum Bounding rectangle is created by using the following equations:

$$xS = xS(tl) = \min i \{oi.xS(tl)\}$$

$$xE = xE(tl) = \max i \{oi.xE(tl)\}$$

$$vS = \min i \{oi.vS\}$$

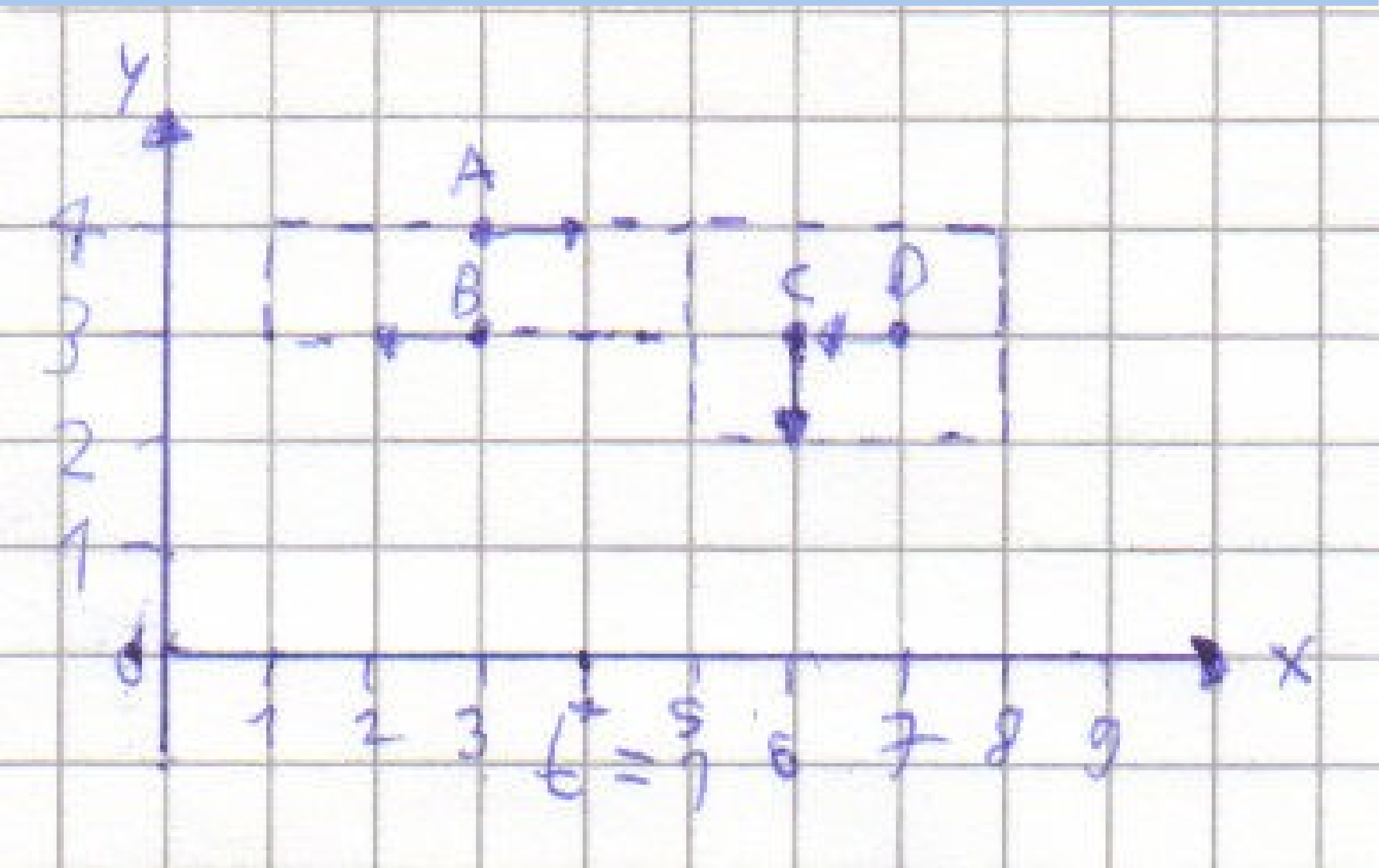
$$vE = \max i \{oi.vE\}$$

TPR-Tree example

	0	0
$[x_s; x_E]$	$[2; 4]$	$[6; 8]$
$[y_s; y_E]$	$[3; 4]$	$[3; 4]$
$[x_{s,v}; x_{E,v}]$	$[-1; 1]$	$[-1; 0]$
$[y_{s,v}; y_{E,v}]$	$[0; 0]$	$[-1; 0]$
$[x_s(t_{ref}); x_E(t_{ref})]$	$[2; 4]$	$[6; 8]$
$[y_s(t_{ref}); y_E(t_{ref})]$	$[3; 4]$	$[3; 4]$
t_{ref}	0	0

$[x; y]$	$[2; 4]$	$[4; 3]$	$[6; 4]$	$[8; 3]$
$[v_x; v_y]$	$[1; 0]$	$[-1; 0]$	$[0; -1]$	$[-1; 0]$
$[x(t_{ref}); y(t_{ref})]$	$[2; 4]$	$[4; 3]$	$[6; 4]$	$[8; 3]$
Name	A	B	C	D

TPR-Tree example



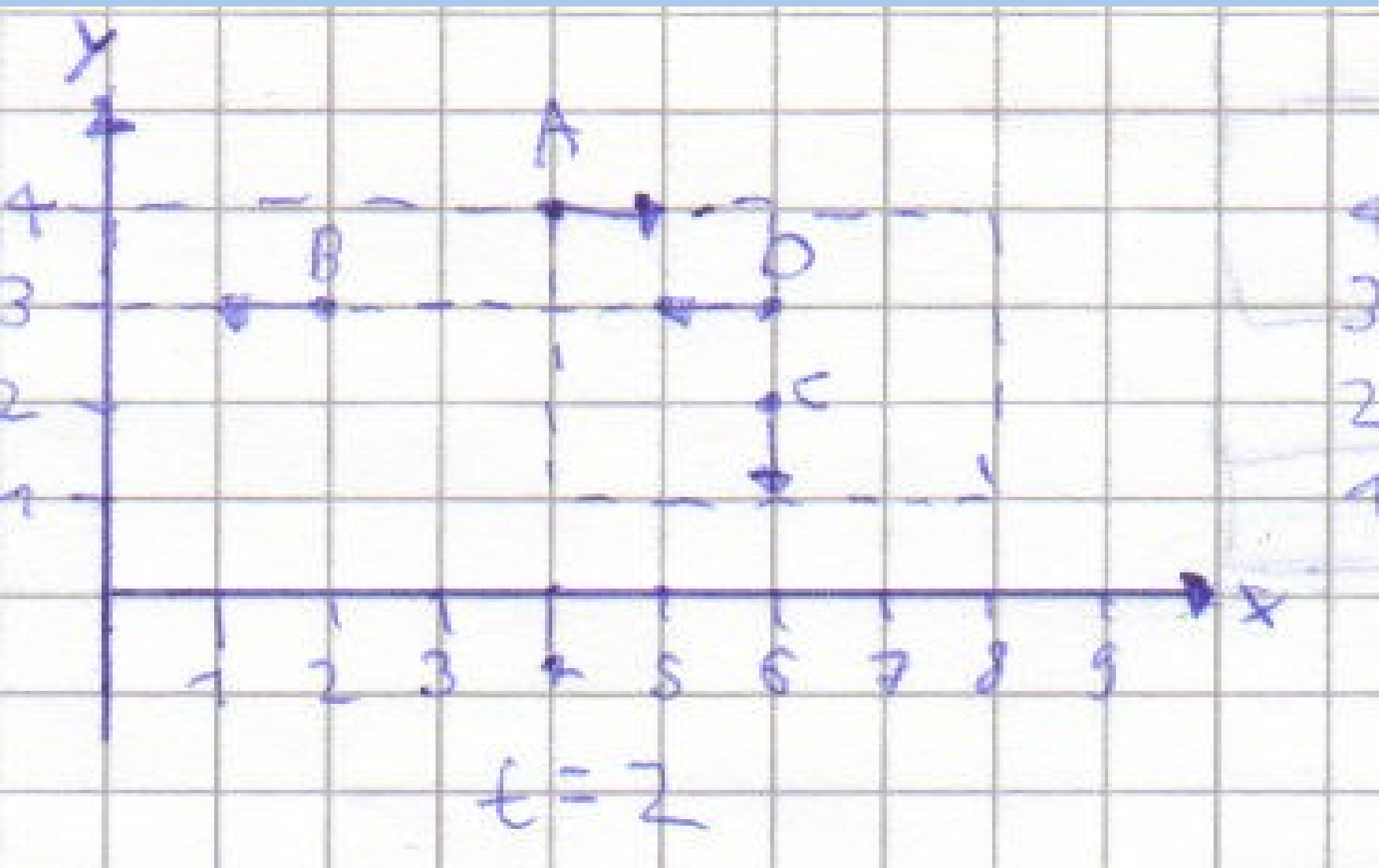
TPR-Tree example

	1	1
$[x_S; x_E]$	[1 ; 5]	[5 ; 8]
$[y_S; y_E]$	[3 ; 4]	[2 ; 4]
$[x_{S.v}; x_{E.v}]$	[-1; 1]	[-1; 0]
$[y_{S.v}; y_{E.v}]$	[0; 0]	[-1; 0]
$[x_S(t_{ref}); x_E(t_{ref})]$	[2 ; 4]	[6 ; 8]
$[y_S(t_{ref}); y_E(t_{ref})]$	[3 ; 4]	[3 ; 4]
t_{ref}	0	0

$[x; y]$	$[3; 4]$	$[3; 3]$		
$[v_x; v_y]$	$[1; 0]$	$[-1; 0]$		
$[x(\text{tree}); y(\text{tree})]$	$[2; 4]$	$[4; 3]$		
Name	A	B		

$[6; 3]$	$[7; 3]$		
$[0; -1]$	$[-1; 0]$		
$[6; 4]$	$[8; 3]$		
C	D		

TPR-Tree example



TPR-Tree example

	2	2		
$[x_S; x_E]$	$[0; 6]$	$[4; 8]$		
$[y_S; y_E]$	$[3; 4]$	$[1; 4]$		
$[x_{S,v}; x_{E,v}]$	$[-1; 1]$	$[-1; 0]$		
$[y_{S,v}; y_{E,v}]$	$[0; 0]$	$[-1; 0]$		
$[x_S(t_{ref}); x_E(t_{ref})]$	$[2; 4]$	$[6; 8]$		
$[y_S(t_{ref}); y_E(t_{ref})]$	$[3; 4]$	$[3; 4]$		
t_{ref}	0	0		

$[x; y]$	$[4; 4]$	$[2; 3]$			$[6; 2]$	$[6; 3]$		
$[v_x; v_y]$	$[1; 0]$	$[-1; 0]$			$[0; -1]$	$[-1; 0]$		
$[x(t_{ref}); y(t_{ref})]$	$[2; 4]$	$[4; 3]$			$[6; 4]$	$[2; 3]$		
Name	A	B			C	D		

Comparison R-Tree and TPR-Tree

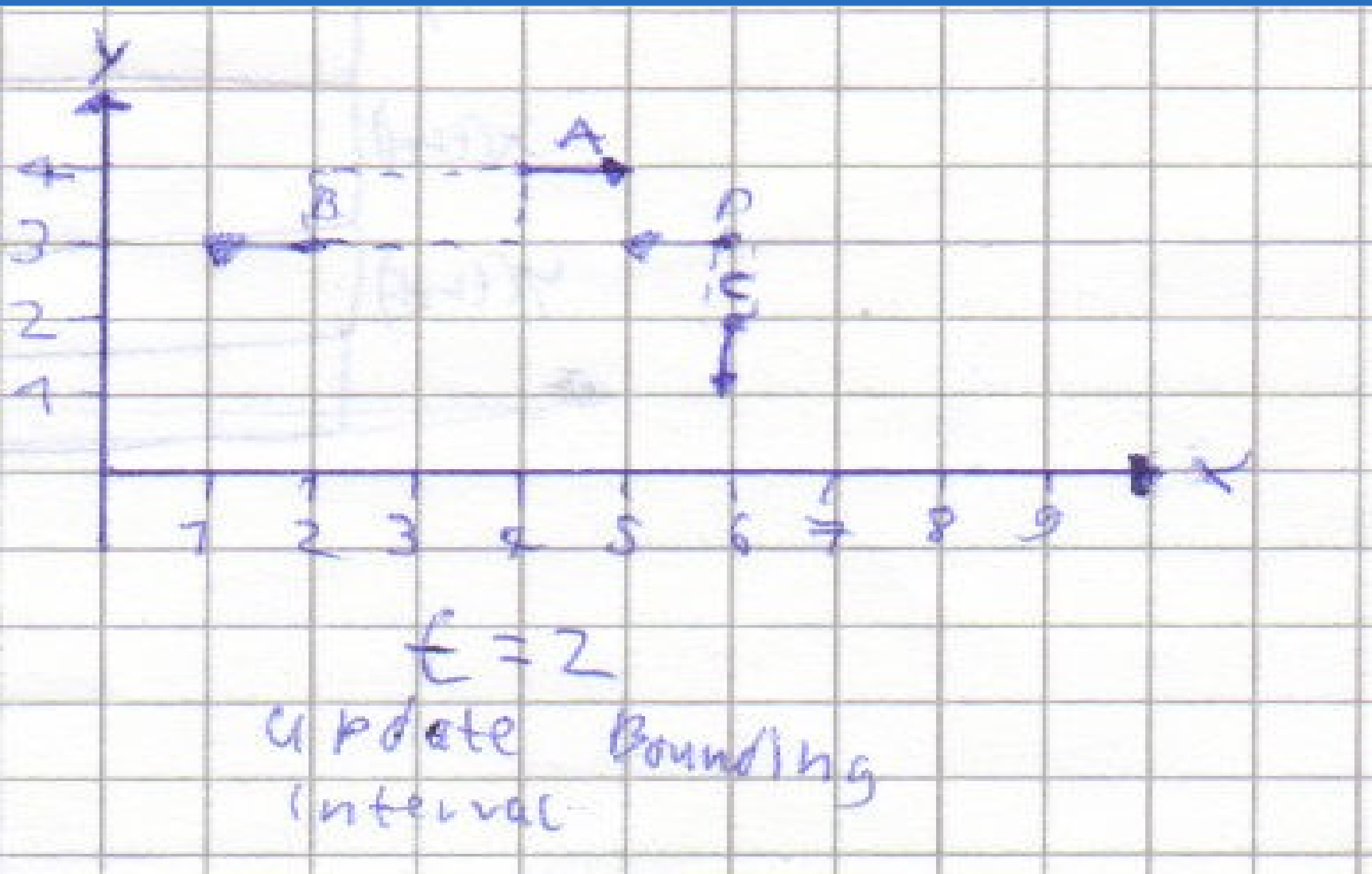
R - Tree

- Static
- Inefficient with moving objects
- Objects positions have to be saved at any time => too much acces to the database
- No sophisticated Query-Algorithms

TPR - Tree

- Dynamic
- Efficient with moving objects
- Objects movings are saved => less frequent database accesses
- Performance lost due to growing rectanlges
- Sophisticated Query-Algorithms

Update of the rectangles



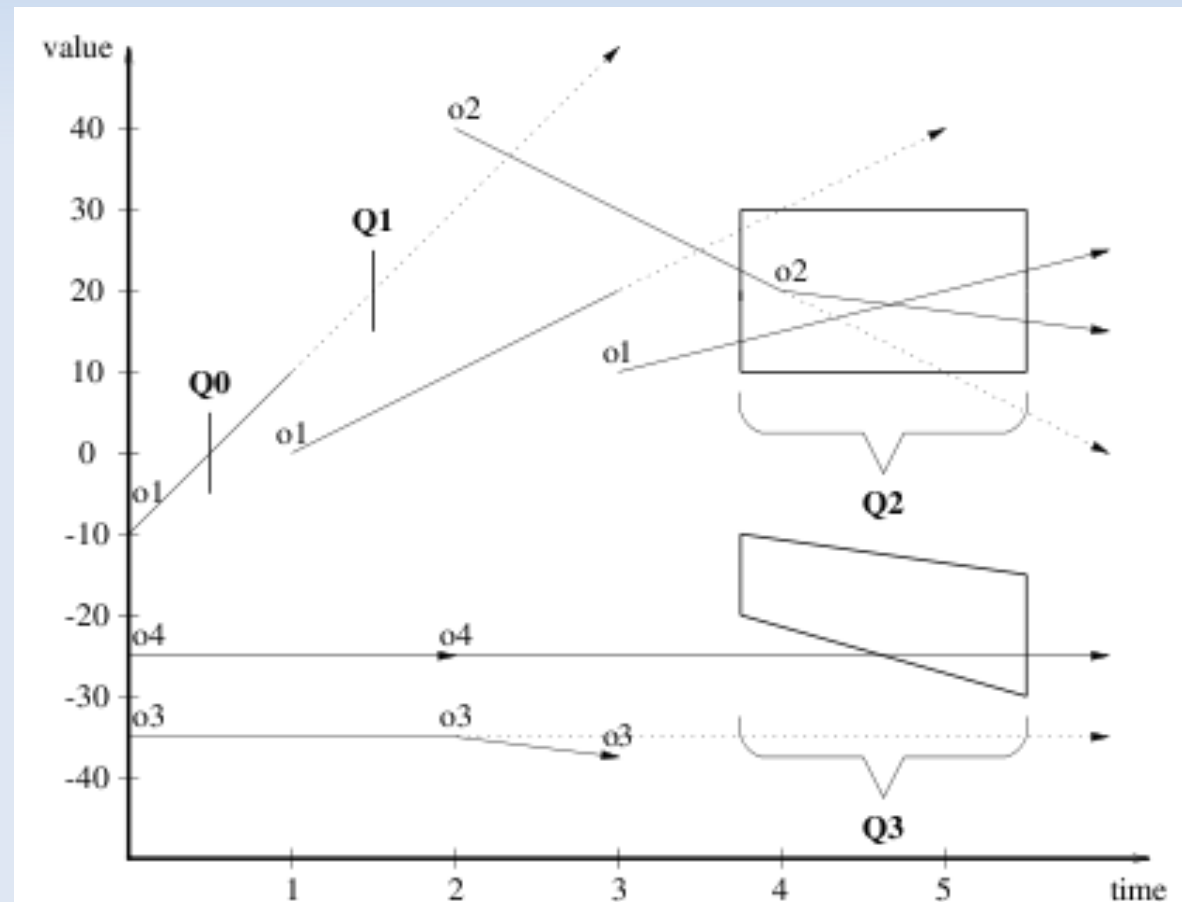
Update of the rectangles

	2	2			
$[x_s; x_E]$	$[2; 4]$	$[6; 6]$			
$[y_s; y_E]$	$[3; 4]$	$[2; 3]$			
$[x_{s,v}; x_{E,v}]$	$[-1; 1]$	$[-1; 0]$			
$[y_{s,v}; y_{E,v}]$	$[0; 0]$	$[-1; 0]$			
$[x_s(t_{ref}); x_E(t_{ref})]$	$[2; 4]$	$[6; 8]$			
$[y_s(t_{ref}); y_E(t_{ref})]$	$[3; 4]$	$[3; 4]$			
t_{ref}	0	0			

$[x; y]$	$[4; 4]$	$[2; 3]$				$[6; 2]$	$[6; 3]$
$[v_x; v_y]$	$[1; 0]$	$[-1; 0]$				$[0; -1]$	$[1; 0]$
$[x(\text{tree}); y(\text{tree})]$	$[2; 4]$	$[4; 3]$				$[6; 4]$	$[8; 3]$
Name	A	B				C	D

Query Types

- Type 1 timeslice query:
 $Q = (R, t)$
- Type 2 window query:
 $Q = (R, tS, tE)$
- Type 3 moving query:
 $Q = (R1, R2, tS, tE)$

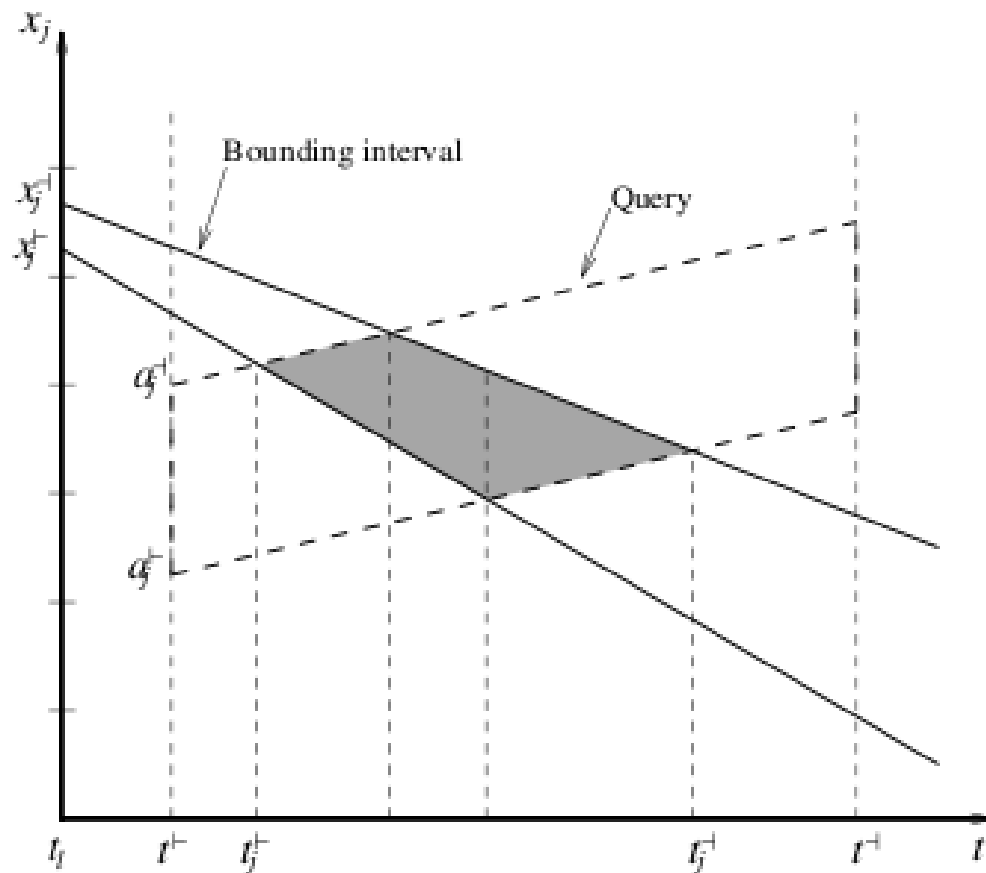


Querying

- A timeslice query is executed just as in a normal R-Tree
- The only difference is that the position of the objects have to be calculated for the specific time of the timeslice query

Querying

- A moving and a window Query works by finding the intersection area of the query



Querying

- The definition of a rectangle R is given by:
 $(xS1, xE1, xS2, xE2, \dots, xSd, xEd, vS1, vE1, vS2, vE2, \dots, vSd, vEd)$
- The Moving Query's definition is $(([aS1, aE1], [aS2, aE2], \dots, [aSd, aEd], [wS1, wE1], [wS2, wE2], \dots, [wSd, wEd]), tS, tE)$
- Having w as velocities of the query. This convention is useful

Querying

- For each dimension j ($j = 1, 2, 3$) the time interval is sought, in which the rectangles overlap $I_j = [t_{Sj}, t_{Ej}]$

I_j is nothing if $a_{Sj} > x_{Ej}(t_S)$ and $a_{Sj}(t_E) > x_{Ej}(t_E)$
or $a_{Ej} < x_{Sj}(t_S)$ and $a_{Ej}(t_E) < x_{Sj}(t_E)$
otherwise I_j is equal to $[t_{Sj}, t_{Ej}]$

First expression: Q is always above R

Second expression: Q is always below R

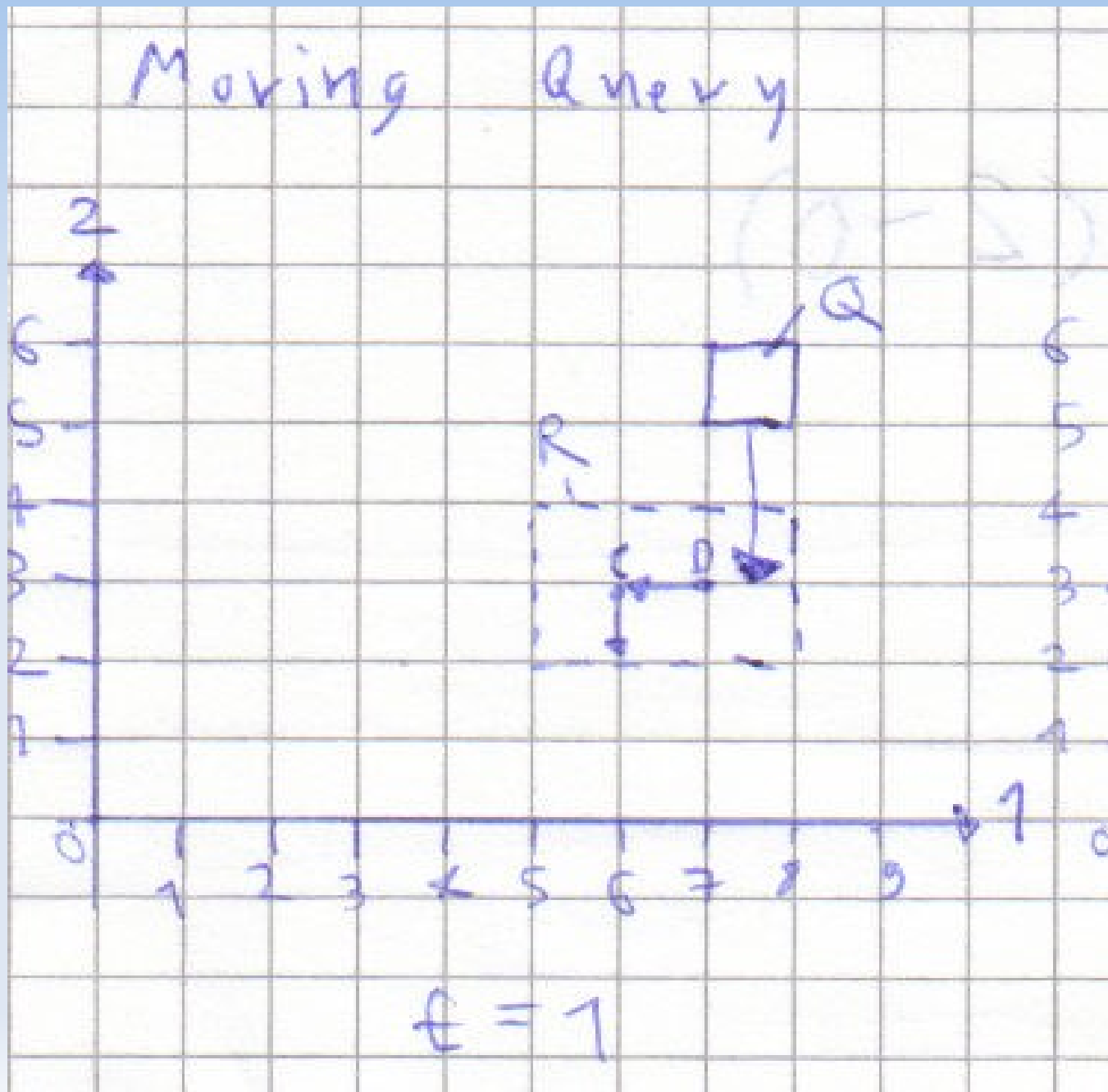
Querying

- t_{Sj} is

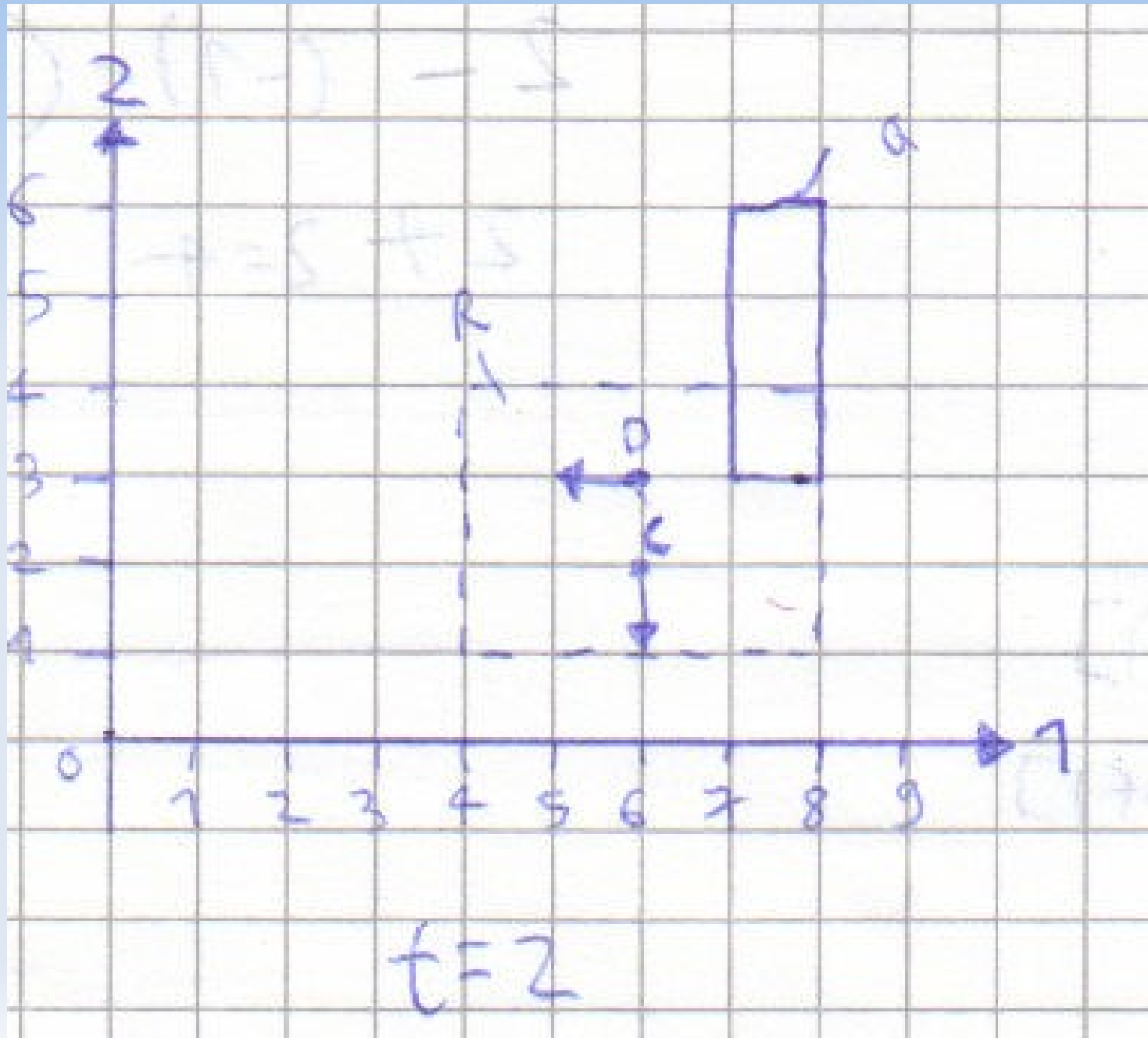
$$\begin{aligned}
 & t_S + (x_{Ej}(t_S) - a_{Sj}) / (w_{Sj} - v_{Ej}) \text{ if } a_{Sj} > x_{Ej}(t_S) \\
 & t_S + (x_{Sj}(t_S) - a_{Ej}) / (w_{Ej} - v_{Sj}) \text{ if } a_{Ej} < x_{Sj}(t_S) \\
 & t_S \text{ otherwise}
 \end{aligned}$$
- t_{Ej} is

$$\begin{aligned}
 & t_S + (x_{Ej}(t_S) - a_{Sj}) / (w_{Sj} - v_{Ej}) \\
 & \text{if } a_{Sj}(t_E) > x_{Ej}(t_E) \\
 & t_S + (x_{Sj}(t_S) - a_{Ej}) / (w_{Ej} - v_{Sj}) \\
 & \text{if } a_{Ej}(t_E) < x_{Sj}(t_E) \\
 & t_E \text{ otherwise}
 \end{aligned}$$

Querying Example



Querying Example



Querying Example

Moving query:

Rectangle R

$[aS_1(t_s); aE_1(t_s)] [7; 8] [xS_1(t_s); xE_1(t_s)] [5; 8]$

$[aS_2(t_s); aE_2(t_s)] [5; 6] [xS_2(t_s); xE_2(t_s)] [2; 4]$

$[aS_1(t_e); aE_1(t_e)] [7; 8] [xS_1(t_e); xE_1(t_e)] [4; 8]$

$[aS_2(t_e); aE_2(t_e)] [3; 6] [xS_2(t_e); xE_2(t_e)] [1; 4]$

$[wS_1; wE_1] [0; 0] [vS_1, vE_1] [-1; 0]$

$[wS_2; wE_2] [-2; 0] [vS_2, vE_2] [-1; 0]$

$[t_s; t_e] [1; 2]$

Querying Example

Werte einsetzen für Dimension 1
 I_1 is nothing if $7 > 8 \wedge 7 > 8 \vee$
 $8 < 5 \wedge 8 < 4$
 $\Rightarrow I_1 = [t_{s_1}, t_{E_1}]$

Calculation t_{E_7}

if($7 > 8$) = wrong
if($8 < 4$) = wrong

$\Rightarrow t_{E_7}$ must be
equal to t_E
 $\Rightarrow t_{E_1} = 2$

Berechnung t_{s_1}

if($7 > 8$) = falsch
if($8 < 5$) = falsch

$\Rightarrow t_{s_1}$ must be
equal to t_s
 $\Rightarrow t_{s_1} = 1$

Finally we get $I_1 = [1, 2]$

Querying Example

Calculation for Dimension 2

I_2 is nothing if $5 > 4 \wedge 3 > 4 \vee$
 $6 < 2 \wedge 6 < 7$

$$\Rightarrow I_2 = [ts_2, tE_2]$$

Calculation tE_2

if $(3 > 4) = \text{wrong}$
if $(6 < 7) = \text{wrong}$

$\Rightarrow tE_2$ must equal
 $t_0 - tE$
 $\Rightarrow tE_2 = 2$

Calculation ts_2

if $(5 > 4) = \text{true}$

$$ts_2 = ts + \frac{x E_2(ts) - a S_2(ts)}{ws_2 - v E_2}$$

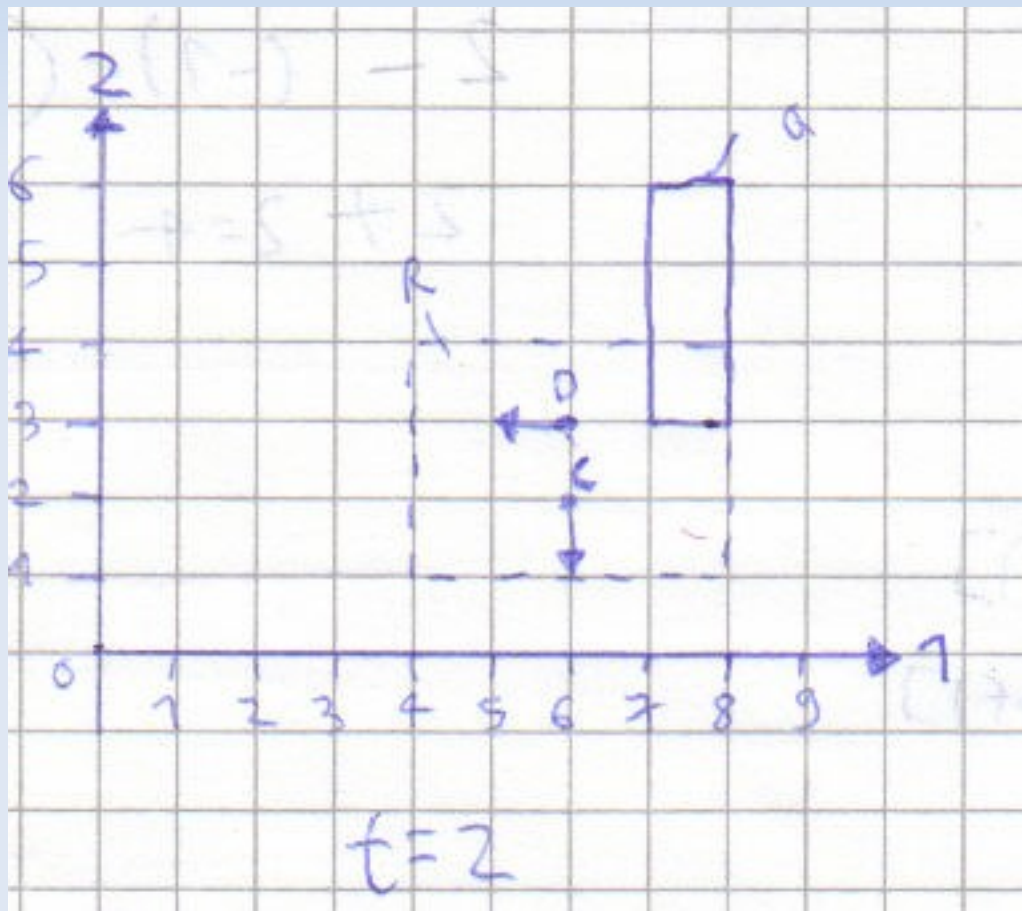
$$ts_2 = 1 + \frac{4 - 5}{-2 - 0} = \frac{3}{2}$$

Calculation of I

I is the intersection of
 I_1 and I_2
 $[1, 2]$ and $[\frac{3}{2}, 2]$
 $\Rightarrow I$ is equal to $[\frac{3}{2}, 2]$

Querying Example

Having $I = [3/2 ; 2]$ as intersection area, we only have to check at T equal 2 since in this case only unbrokenly numbers of T are wished. We need to look for an Object contained in Q at $T=2$



Performance experiments

- 1'000*1'000 Kilometer area
- 100'000 moving objects
- 20 destinations
- 3 different speeds
- 1/6 acceleration, 2/3 Constant speed, 1/6 slow down
- UI means Update Interval
- UI usually = 60, this means there are 60 time periods between two updates

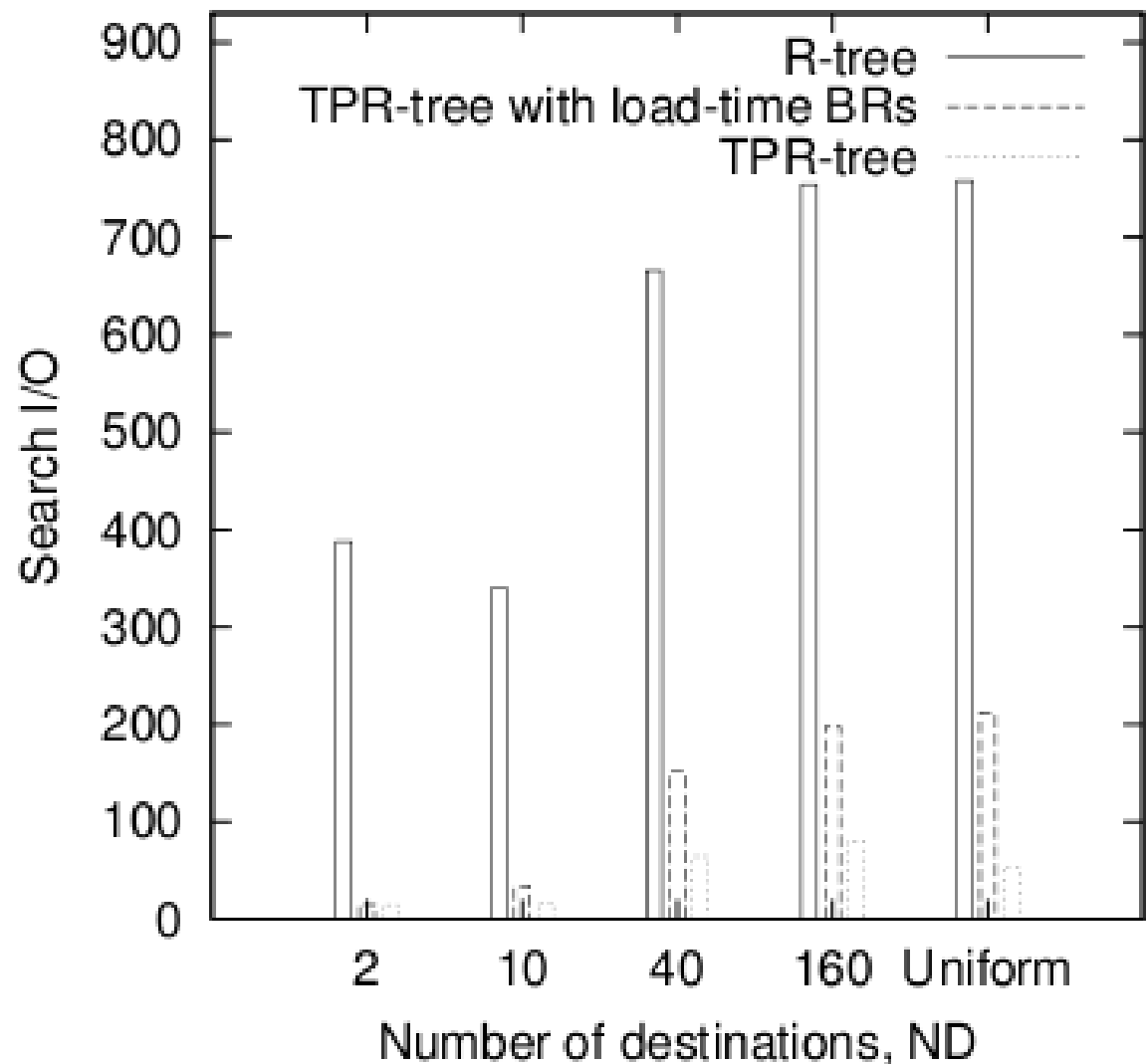
Performance experiments

Search

Performance for
varying Number
of destinations

Expectation:

TPR-Tree < TPR-
Tree with load
time Brs << R-
Tree



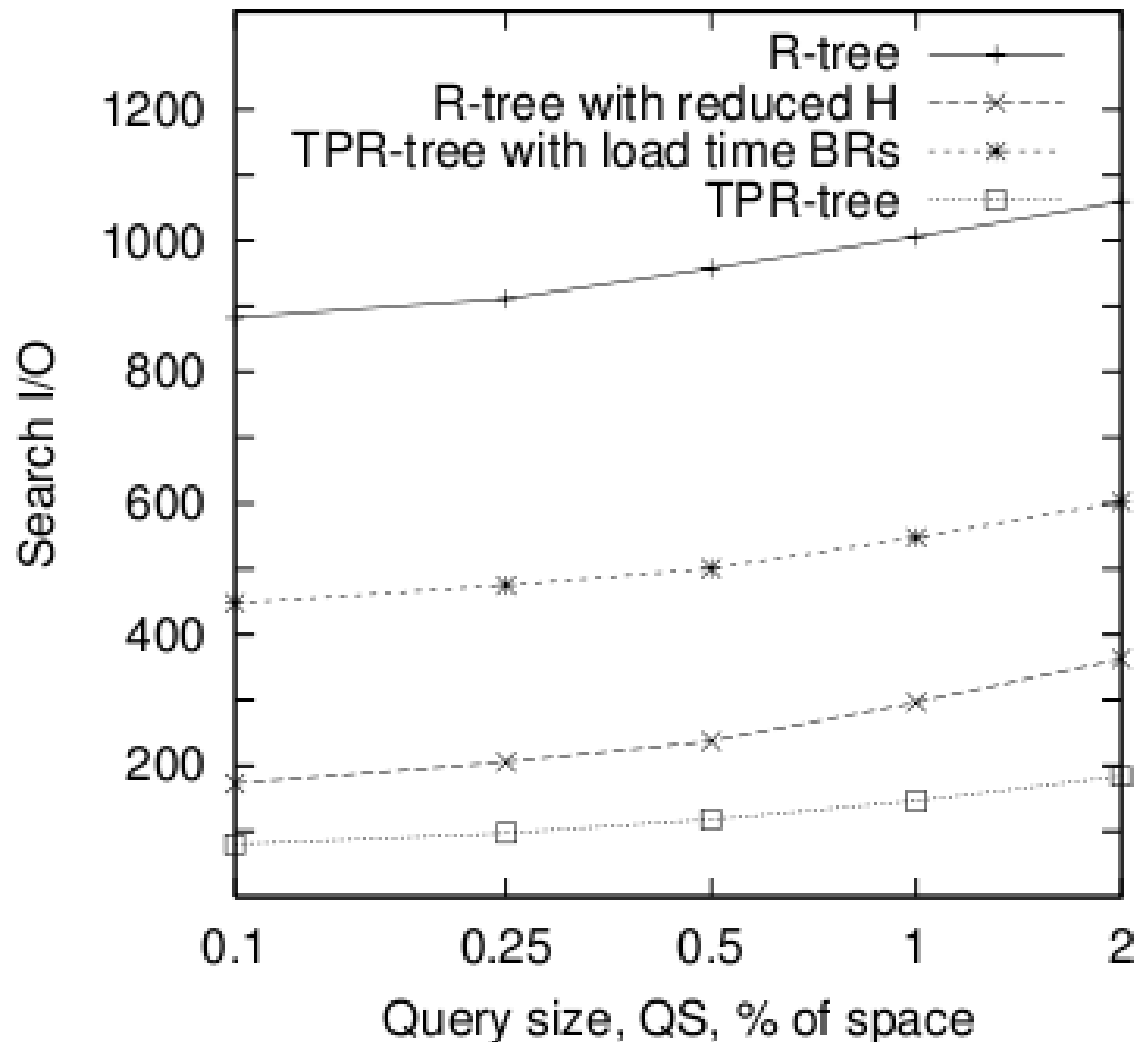
Performance experiments

Search

Performance for
varying size of
Query

Expectation:

TPR-Tree < TPR-
Tree with load
time Brs << R-
Tree



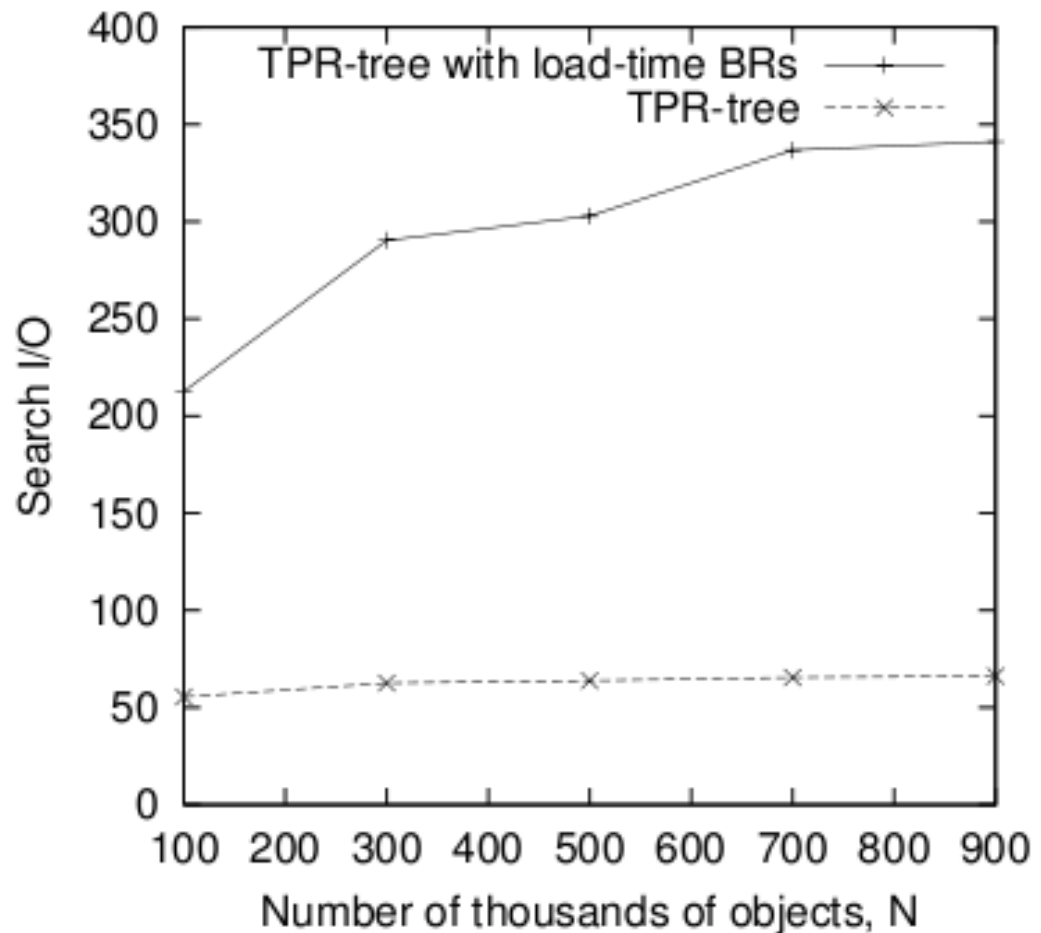
Performance experiments

Search

Performance for
varying number
of objects

Expectation:
TPR-Tree < TPR-
Tree with load
time Brs << R-
Tree

- R-Tree has far to much Search I/Os

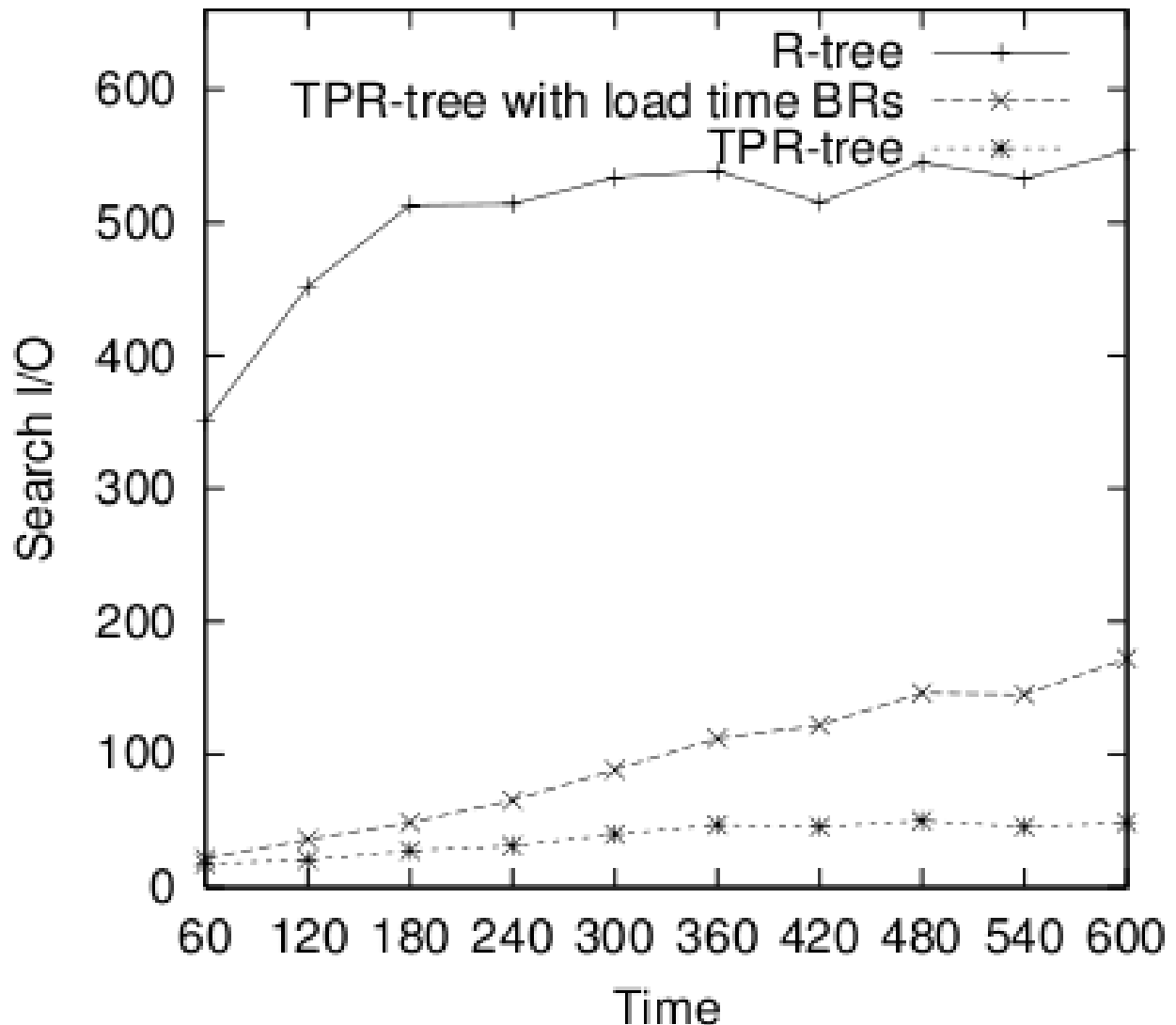


Performance experiments

Search
Performance
over growing time

Expectation:
 $\text{TPR-Tree} < \text{TPR-Tree with load time BRs} < \text{R-Tree}$

=> Load time
BRs keep
growing over the
time

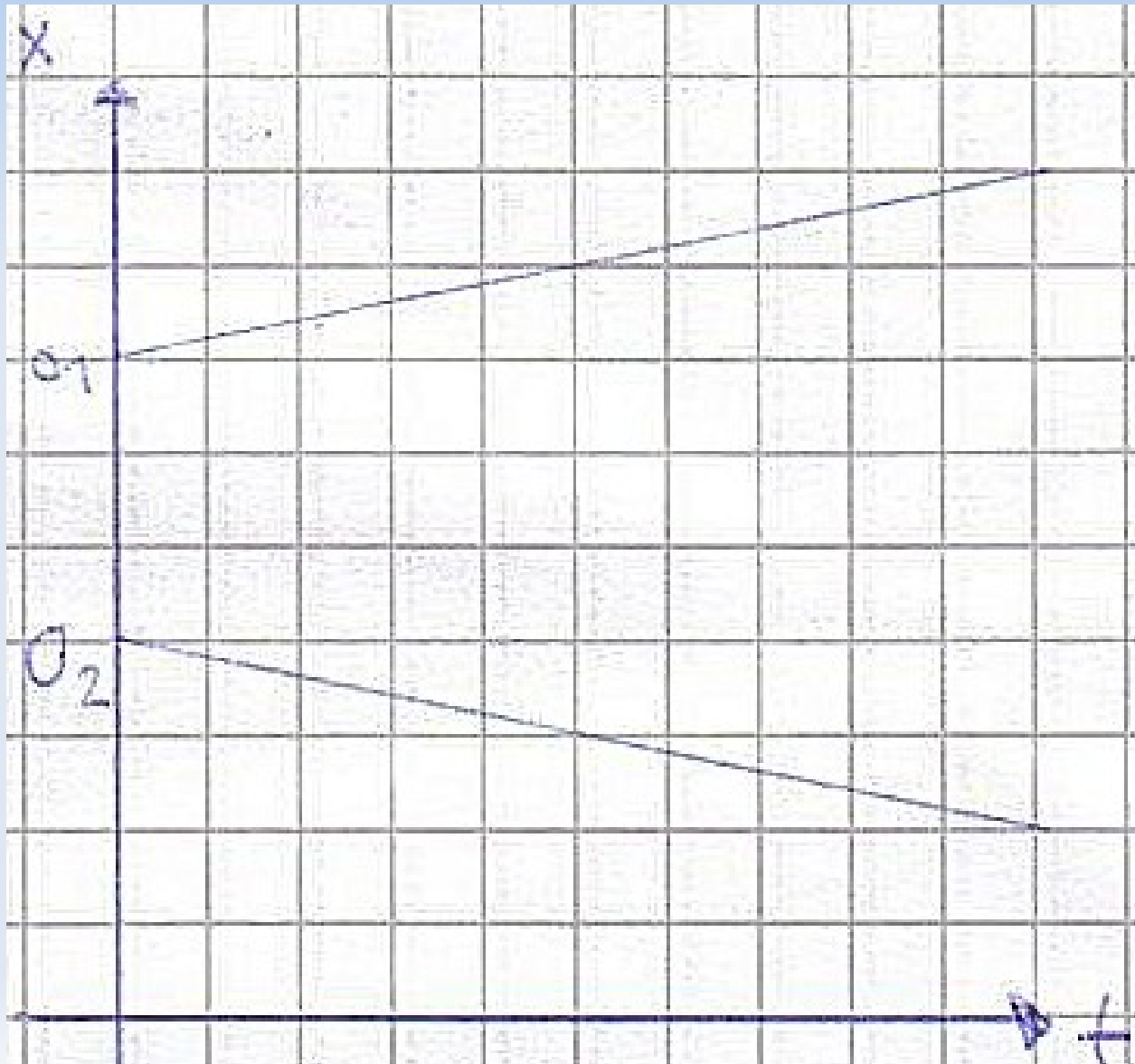


Heuristics for tree Organisation

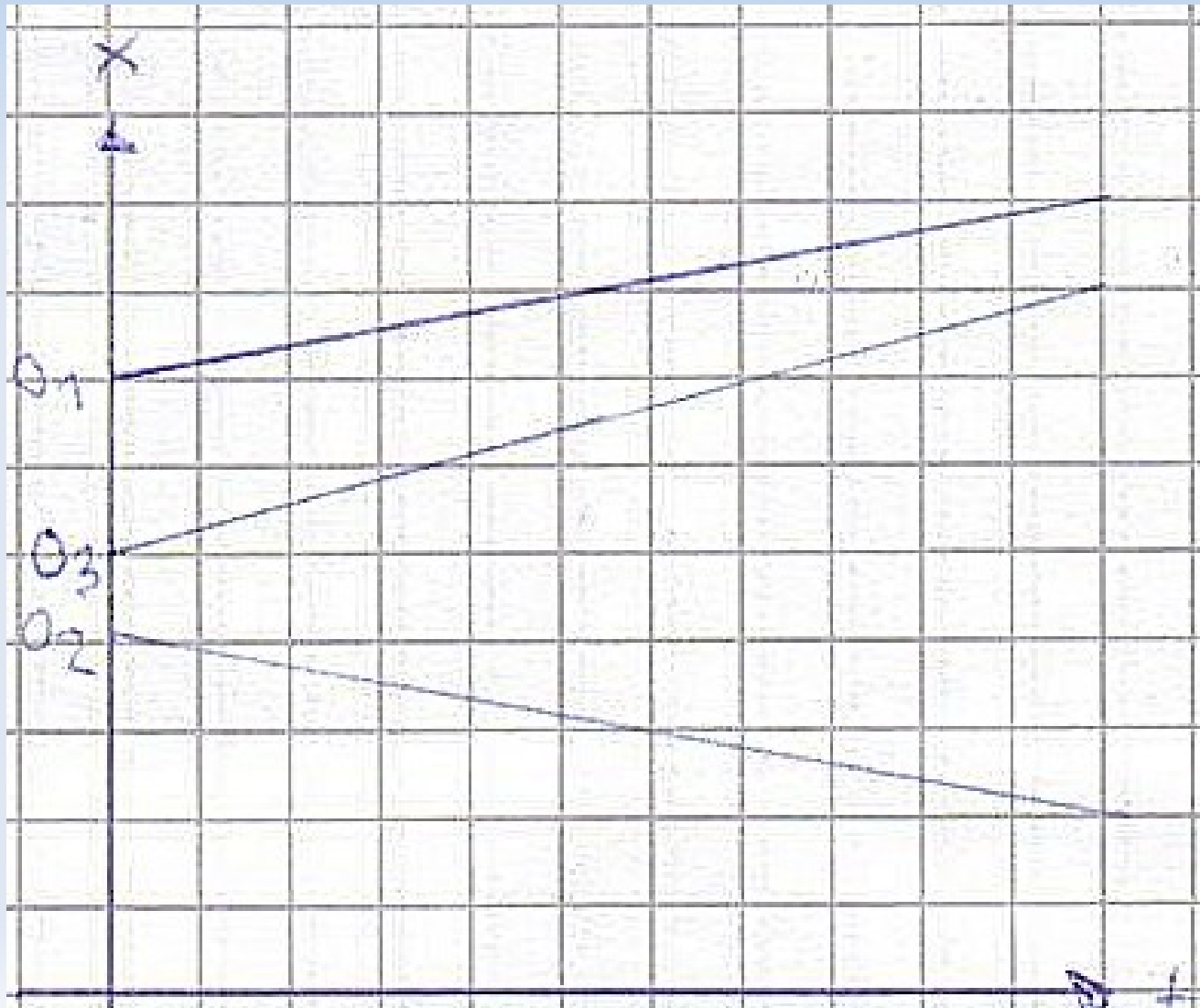
- The TPR-Tree uses insertion and split algorithms optimised to minimize the area, the perimeters and the overlap of bounding rectangles.
- The functions are integrals over the time

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

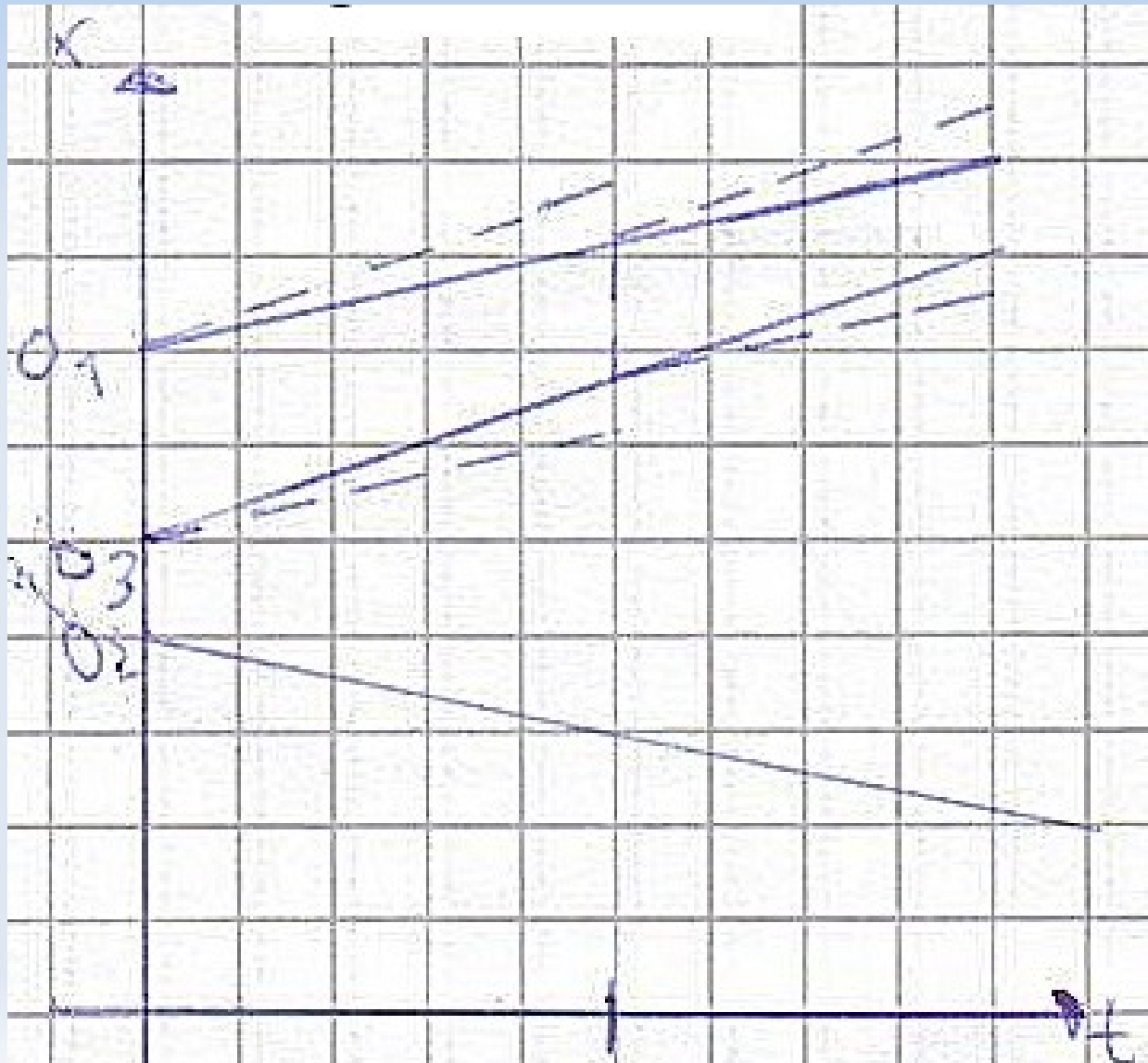
Examples: Insertion



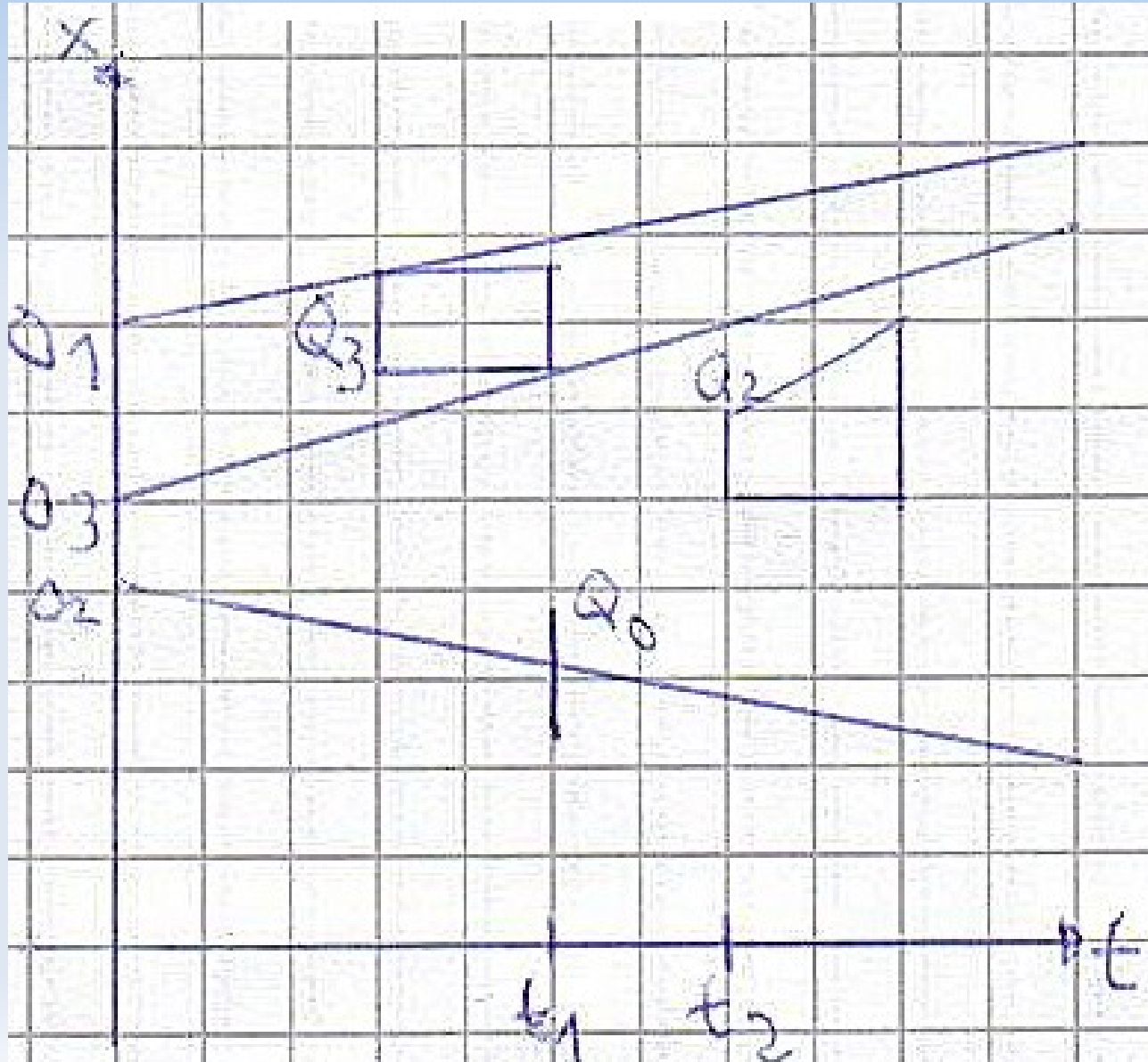
Examples: Insertion



Examples: Minimum Bounding Rectangle



Examples: Querying



Examples: Deletion

