

# Orthogonal Range Searching in 2D using Ball Inheritance

Mads Ravn

Computer Science, Aarhus University

2015

- Giv en præsentation af den i specialet introducerede simplificerede datastruktur til range searching i 2d. (3)
- Beskriv ball-inheritance problemet og forklar sammenhængen til range searching. (2)
- Beskriv også det klassiske kd-træ (1)
- og fortæl om hvilke eksperimenter du har foretaget for at sammenligne performance af de to strukturer. Forklar hvad du så og om det var som forventet. (4)

# Outline

## 1 Introduction

- Orthogonal Range Searching
- Previous data structures

## 2 Ball Inheritance Search

- Ball Inheritance Problem
- Ball Inheritance Search Data Structure

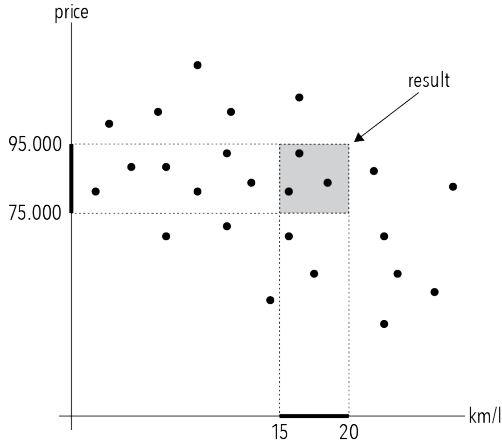
### 3 Resultater

- Resultater

# Outline

- 1 Introduction
  - Orthogonal Range Searching
  - Previous data structures
- 2 Ball Inheritance Search
  - Ball Inheritance Problem
  - Ball Inheritance Search Data Structure
- 3 Resultater
  - Resultater

# Orthogonal Range Searching



# Preliminaries

## Orthogonal range searching

- Svar effektivt på forespørgslen  $q = [x_1, x_2] \times [y_1, y_2]$ .
- $p \in [x_1, x_2] \times [y_1, y_2] \Leftrightarrow p_x \in [x_1, x_2] \wedge p_y \in [y_1, y_2]$

## Preliminaries

- $n$  punkter fra  $\mathbb{R}^2$ . Alle koordinater er unikke
- Rank space
- $n$  er en potens af 2
- static og output-sensitive

# Orthogonal Range Searching

Vi er givet  $n$  punkter fra  $\mathbb{R}^2$  som vi ønsker at indsætte i en datastruktur sådan at vi kan svare effektivt på forespørgslen  $q = [x_1, x_2] \times [y_1, y_2]$ . Et punkt  $p = (p_x, p_y)$  ligger i  $q = [x_1, x_2] \times [y_1, y_2]$  hvis  $p_x \in [x_1, x_2]$  og  $p_y \in [y_1, y_2]$ . Man kunne derfor sige at et 2-dimensionelt query består af to 1-dimensional sub-queries. Kommer til at virke for alle tre datastrukturer.

# Outline

## 1 Introduction

- Orthogonal Range Searching
- Previous data structures

## 2 Ball Inheritance Search

- Ball Inheritance Problem
- Ball Inheritance Search Data Structure

## 3 Resultater

- Resultater



# kd-træ

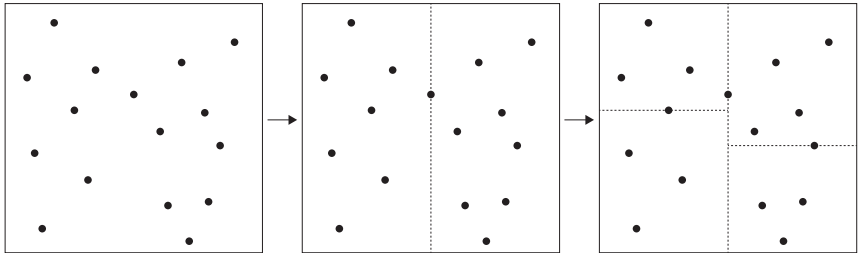
Jon L. Bentley. Multidimensional binary search trees used for associative searching. 1975.

- $\mathcal{O}(n)$  plads
- $\mathcal{O}(\sqrt{n} + k)$  tid

Givet  $n$  punkter: Punkterne bliver sorteret efter  $x$  eller  $y$  på skift. Median bliver fundet og punkterne mindre end medianen bliver givet til venstre barn og punkterne højere end medianen bliver givet til højre barn. Et punkt per blad i træet.

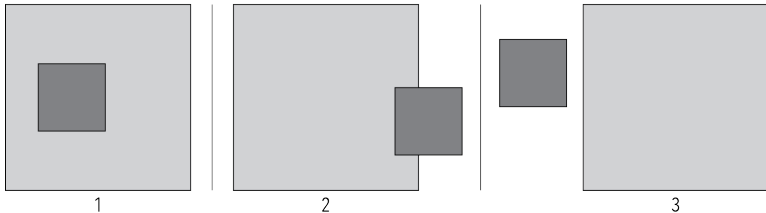
# Opbygning af kd-træ

Det  $\lceil \frac{n}{2} \rceil$ 'te element bliver valgt som median. Dette element fungerer som en skille-linje mellem de to punkt-mængder. Medianen bliver låst fast på denne plads i arrayet. Det er nu skillelinjen mellem de to regioner.

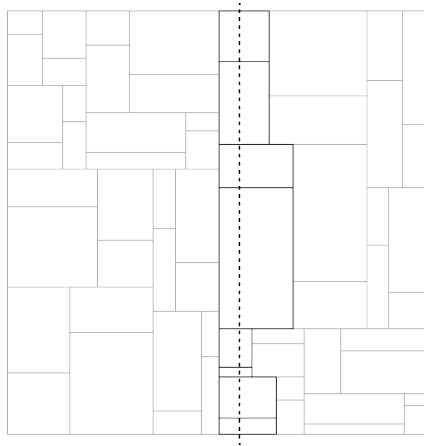


# Søgning i kd-træ

■ = node region  
■ = search query



# Søgning i kd-træ



# BISintro

Ball Inheritance Search (BIS) er en datastruktur bygget som en simplificering af den datastruktur Chan et al laver.

# Outline

- 1 Introduction
  - Orthogonal Range Searching
  - Previous data structures
- 2 Ball Inheritance Search
  - Ball Inheritance Problem
  - Ball Inheritance Search Data Structure
- 3 Resultater
  - Resultater

# Ball Inheritance

- Vi er givet et perfekt binært træ.

# Ball Inheritance

- Vi er givet et perfekt binært træ.
- Roden indeholder  $n$  punkter(bolde) som er blevet fordelt sådan at hvert blad lagrer et punkt. Elementerne i roden er sorteret.



# Ball Inheritance

- Vi er givet et perfekt binært træ.
- Roden indeholder  $n$  punkter(bolde) som er blevet fordelt sådan at hvert blad lagrer et punkt. Elementerne i roden er sorteret.
- Hver knude har en liste over hvilke bolder der går igennem den. De bolde ender så i træet med rod i den knude. Boldene i knudens liste har samme rækkefølge som boldene i forældre-knudens liste.

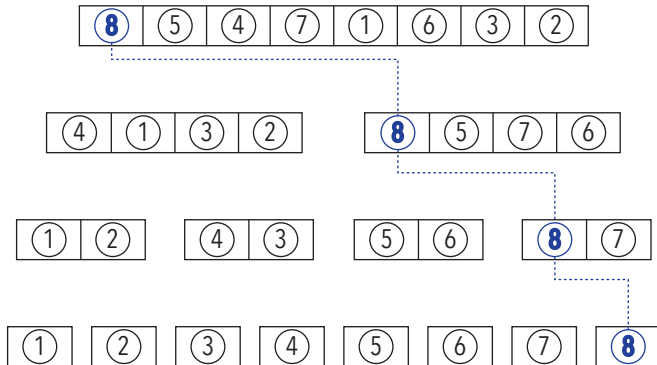
# Ball Inheritance

- Vi er givet et perfekt binært træ.
- Roden indeholder  $n$  punkter(bolde) som er blevet fordelt sådan at hvert blad lagrer et punkt. Elementerne i roden er sorteret.
- Hver knude har en liste over hvilke bolder der går igennem den. De bolde ender så i træet med rod i den knude. Boldene i knudens liste har samme rækkefølge som boldene i forældre-knudens liste.
- Løs: Givet en knude og et index i knudens liste, hvilket blad ender denne bold ved? Vi kan følge bolden

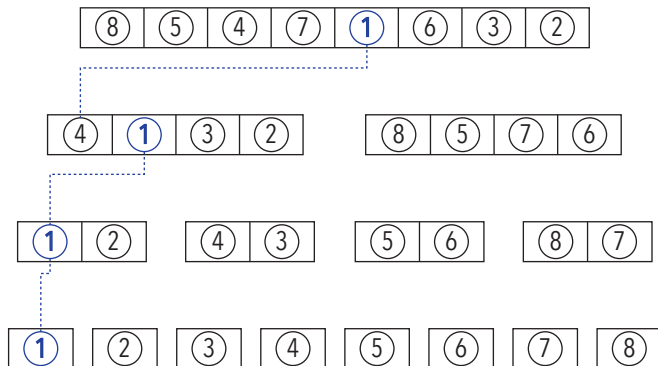
# Ball Inheritance

- Vi er givet et perfekt binært træ.
- Roden indeholder  $n$  punkter(bolde) som er blevet fordelt sådan at hvert blad lagrer et punkt. Elementerne i roden er sorteret.
- Hver knude har en liste over hvilke bolder der går igennem den. De bolde ender så i træet med rod i den knude. Boldene i knudens liste har samme rækkefølge som boldene i forældre-knudens liste.
- Løs: Givet en knude og et index i knudens liste, hvilket blad ender denne bold ved? Vi kan følge bolden
- Vi kan nu følge en bold fra en knude til et blad med  $\mathcal{O}(\lg n)$  skridt.

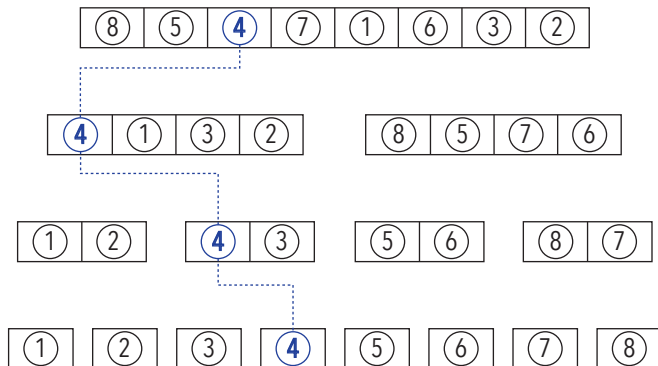
# ball inheritance



# ball inheritance



# ball inheritance



# RANK SELECT

Husk RANK-SELECT struktur. Hvordan fungerer den? Med  $\lg n$  og  $\lg \Sigma$ .

# ball inheritance

1	1	0	1	0	1	0	0
---	---	---	---	---	---	---	---

1	0	1	0
---	---	---	---

1	0	1	0
---	---	---	---

0	1
---	---

1	0
---	---

0	1
---	---

1	0
---	---

1	2	3	4	5	6	7	8
---	---	---	---	---	---	---	---



# Faster Queries

Vi ønsker at gøre antallet skridt fra en knude til et blad mindre. Vi udvider alfabetet på udvalgte niveauer. Det bruger

- $\mathcal{O}(\frac{n}{\epsilon}) = \mathcal{O}(n)$  plads
- $\mathcal{O}(\lg^\epsilon n)$  tid

hvor  $\epsilon > 0$  er en arbitrær lille konstant. Space-time tradeoff. Vis koncept, tid og plads her

# Outline

## 1 Introduction

- Orthogonal Range Searching
- Previous data structures

## 2 Ball Inheritance Search

- Ball Inheritance Problem
- Ball Inheritance Search Data Structure

### 3 Resultater

- Resultater

# Ball Inheritance Search

- $\mathcal{O}(n)$  plads
- $\mathcal{O}(\lg n + k \cdot \lg^\epsilon n)$  tid, hvor  $\epsilon > 0$  er en arbitrær lille konstant

## x-range

- Vi oversætter vores query til rank space  
 $[x_1, x_2] \times [y_1, y_2] \Rightarrow [\hat{x}_1, \hat{x}_2] \times [\hat{y}_1, \hat{y}_2]$ .

## x-range

- Vi oversætter vores query til rank space  
 $[x_1, x_2] \times [y_1, y_2] \Rightarrow [\hat{x}_1, \hat{x}_2] \times [\hat{y}_1, \hat{y}_2]$ .
- Vi går ned til least common ancestor af  $\hat{x}_1$  og  $\hat{x}_2$  og herfra ned til  $\hat{x}_1$  og  $\hat{x}_2$ . På den måde finder vi knuder der kun indeholder punkter i  $[x_1, x_2]$ .

## y-range

- Vi har opdateret  $[\hat{y}_1, \hat{y}_2]$  fra roden til både  $\hat{x}_1$  og  $\hat{x}_2$ . Dvs vi ved hvilke bolde i hver knude vi fandt før der indeholder punkter i  $[y_1, y_2]$ .

## y-range

- Vi har opdateret  $[\hat{y}_1, \hat{y}_2]$  fra roden til både  $\hat{x}_1$  og  $\hat{x}_2$ . Dvs vi ved hvilke bolde i hver knude vi fandt før der indeholder punkter i  $[y_1, y_2]$ .
- Vi har nu nogle knuder og lister over indeces i disse knuder. Det er præcis det problem ball inheritance løser. Vi kan nu bruge ball inheritance på alle disse knuder til at finde ud af hvilke blade der indeholder punkter i  $[y_1, y_2]$ .

# ballinheritance

Vi har nu at hver knude der er fully contained laver ball inheritance på det y-range den får givet. Det tager  $\mathcal{O}(k \cdot \lg^\epsilon n)$  tid. Det tager  $\mathcal{O}(\lg n)$  at lave binær søgning og at gå fra roden til  $\hat{x}_1$  og  $\hat{x}_2$ .



# ballinheritance

Vi har nu at hver knude der er fully contained laver ball inheritance på det y-range den får givet. Det tager  $\mathcal{O}(k \cdot \lg^\epsilon n)$  tid. Det tager  $\mathcal{O}(\lg n)$  at lave binær søgning og at gå fra roden til  $\hat{x}_1$  og  $\hat{x}_2$ . Det giver en kørselstid på  $\mathcal{O}(\lg n + k \cdot \lg^\epsilon n)$  for at finde  $k$  punkter.

# plads

Denne datastruktur bruger  $\mathcal{O}(n)$  plads.

- Bit vectors.

# plads

Denne datastruktur bruger  $\mathcal{O}(n)$  plads.

- Bit vectors.
- Store hop (Kommer vi til)

# plads

Denne datastruktur bruger  $\mathcal{O}(n)$  plads.

- Bit vectors.
- Store hop (Kommer vi til)
- Egentlig punkter

# plads

Denne datastruktur bruger  $\mathcal{O}(n)$  plads.

- Bit vectors.
- Store hop (Kommer vi til)
- Egentlig punkter
- Binær søgning

# Små hop

Hvert niveau gemmer  $n$  bits som indikerer om bolden er gået til højre eller venstre. Hvert 32 bit gemmer vi et 32 bit major checkpoint. Precomputed tabel med 16 bit tal som tæller antal 1-entries.  $\mathcal{O}(n)$  bits per level.

# Store hop

$\mathcal{O}(\lg \Sigma)$  per entry.  $\Sigma = 2^{B^i}$ . Så plads er  $\mathcal{O}(B^i)$  bits per entry.

Det er

$$\sum_{i=1}^{\lg_B \lg n} \frac{\lg n}{B^i} \cdot \mathcal{O}(B^i) = \mathcal{O}(\lg n \cdot \lg_B \lg n)$$

for hele kæden. Vælg nu  $B = \Omega(\lg^\epsilon n)$ .

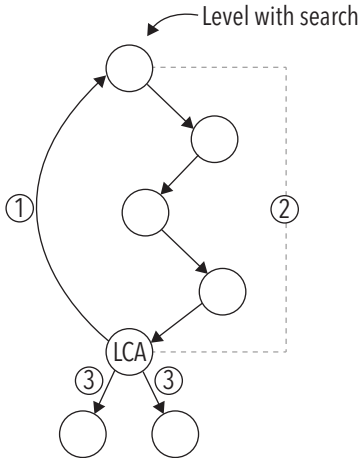
Tiden for de store hop er højst  $\mathcal{O}(B \lg_B \lg n)$ . Vælg  $B = \lg^{\epsilon/2} n = \Omega(\lg \lg n)$ .



# OBIS

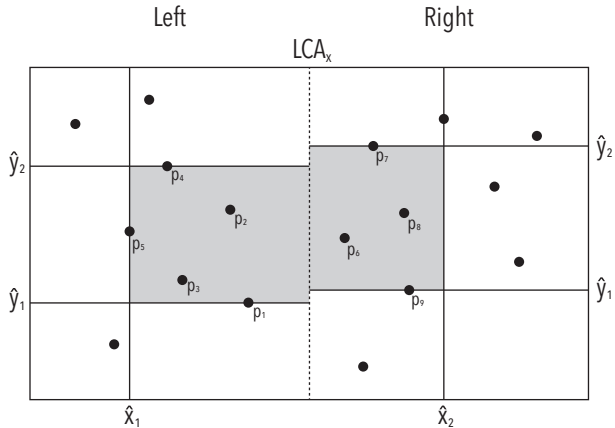
OBIS af Chan et al. Med  $\mathcal{O}(n)$  plads og  $\mathcal{O}(\lg \lg n + k \cdot \lg^\epsilon n)$ .  
Bruger også Ball Inheritance til at finde de  $k$  punkter.

# OBIS



- Op til nærmeste level med pred-search
- Gå ned til LCA højst  $\lg \lg n$  levels nede.
- Gå ned og find resultater i begge børn af LCA.

# OBIS



## 1 Introduction

- ## 2 Ball Inheritance Search

- ### 3 Resultater

- 
- AARHUS  
UNIVERSITY  
DEPARTMENT OF COMPUTER SCIENCE

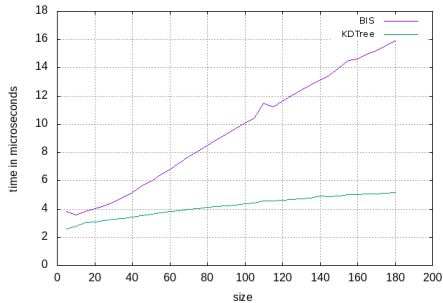
## setup

- Square area  $\sqrt{n} \cdot \sqrt{k} \times \sqrt{n} \cdot \sqrt{k}$  returnerer  $k$  punkter.
- Slices af størrelse  $k$  skulle returnere  $k$  punkter.  
 $[0, n] \times [y, y + k]$
- $\sqrt{n} + k = \lg n + k \cdot \lg^\epsilon n \Leftrightarrow k = \frac{\sqrt{n} - \lg n}{\lg^\epsilon n - 1}$

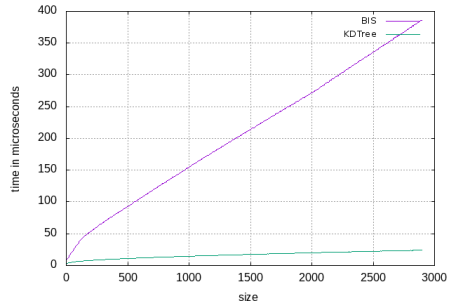


# Squared

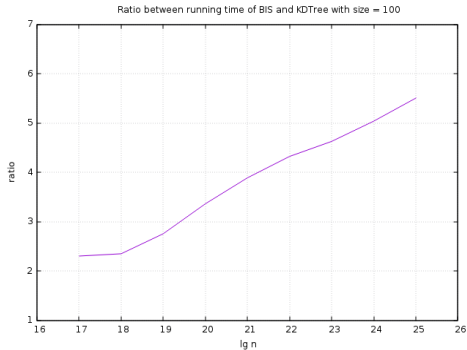
KDTree vs BIS - squared with  $\lg n = 17$



KDTree vs BIS - squared with  $\lg n = 25$



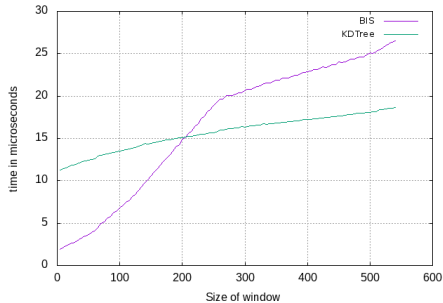
# Squared



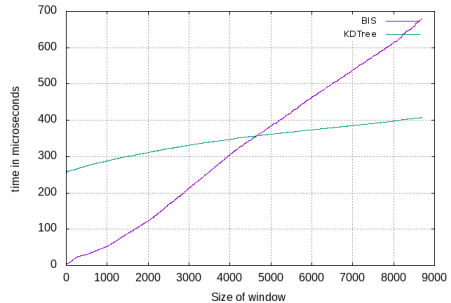


# vertical

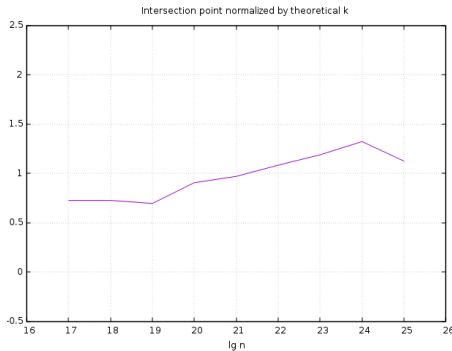
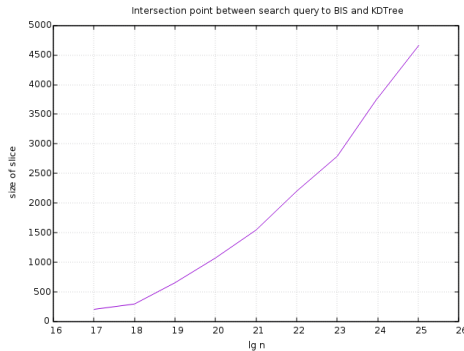
KDTree vs BIS - vertical 17



KDTree vs BIS - vertical 25

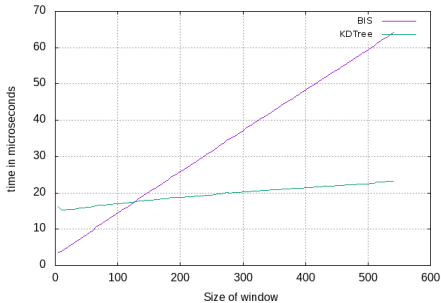


# vertical

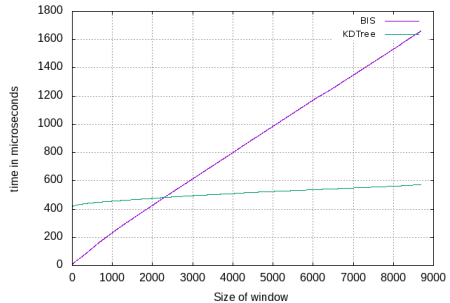


# horizontal

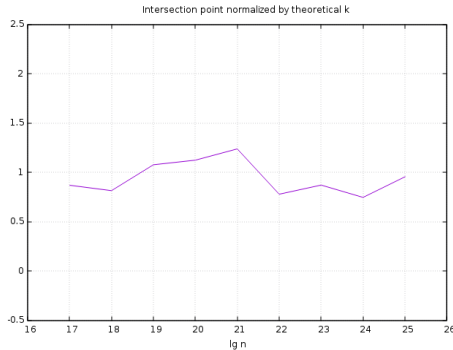
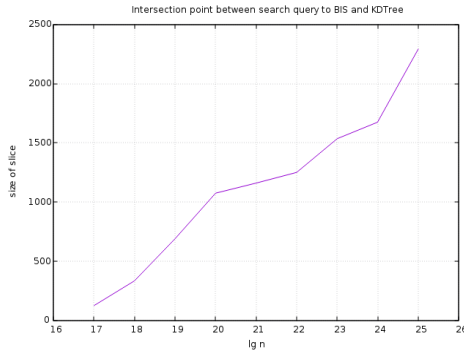
KDTree vs BIS - horizontal 17



KDTree vs BIS - horizontal 25

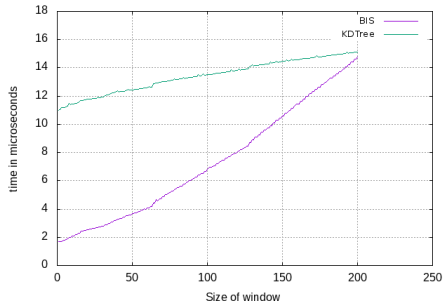


# horizontal

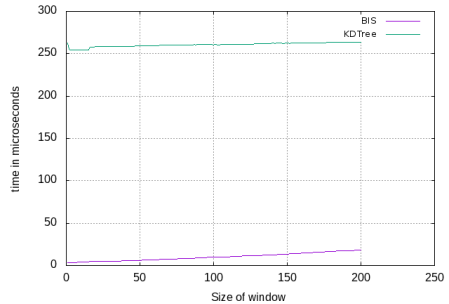


# small k

KDTree vs BIS - vertical 17

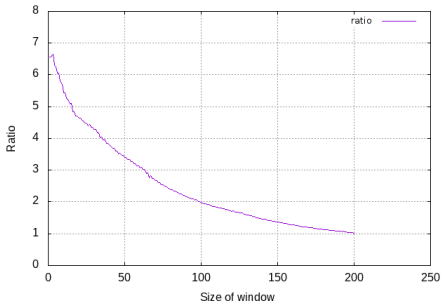


KDTree vs BIS - vertical 25

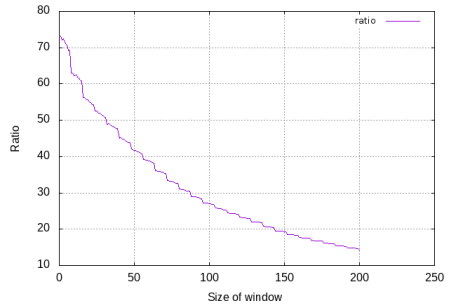


# small k

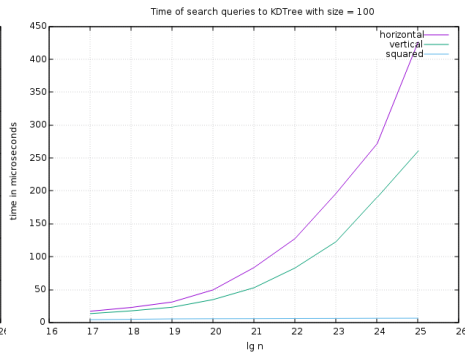
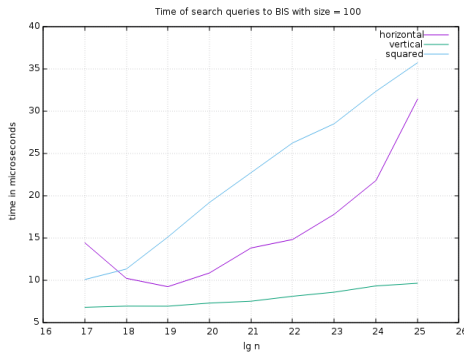
KDTree vs BIS - vertical 17



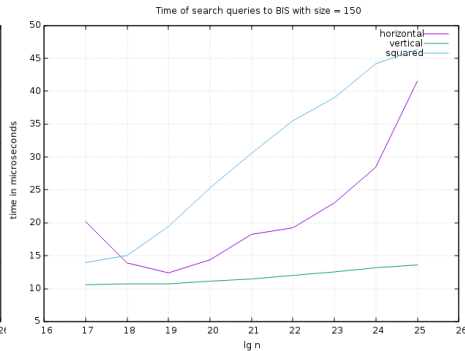
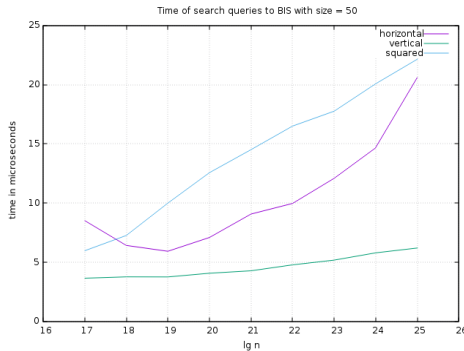
KDTree vs BIS - vertical 25



# small k

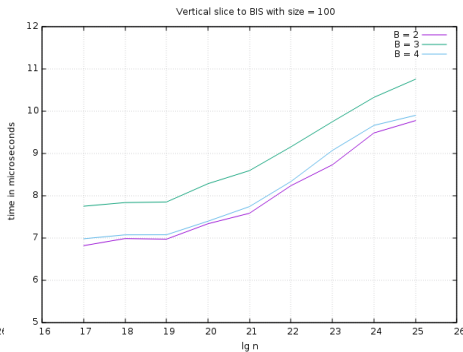
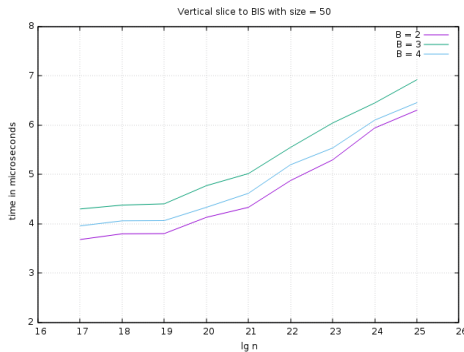


# small k

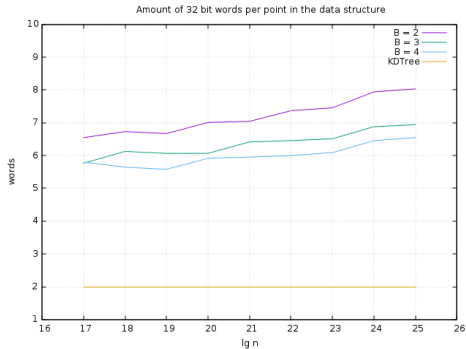




# small k



## sizes



# small k

