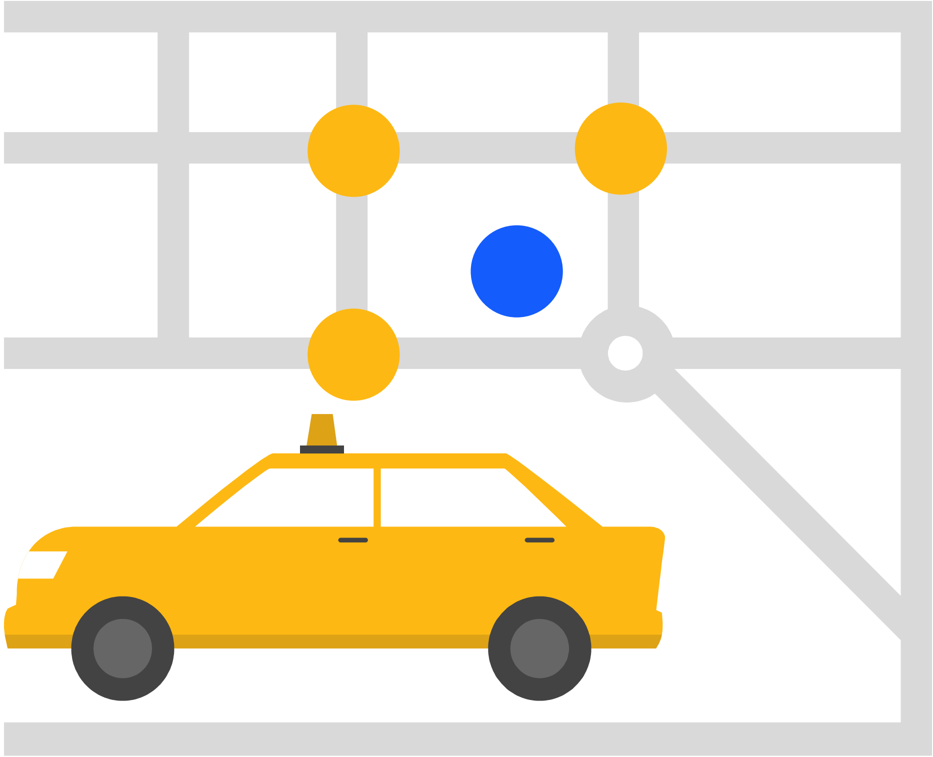
Taxi Trajectory Analysis

EDAA - G04

Diogo Rodrigues
Eduardo Correia
João Sousa

Problem recap

Problem of **mapping GPS coordinates** with **network nodes** that represent the real-life roads.



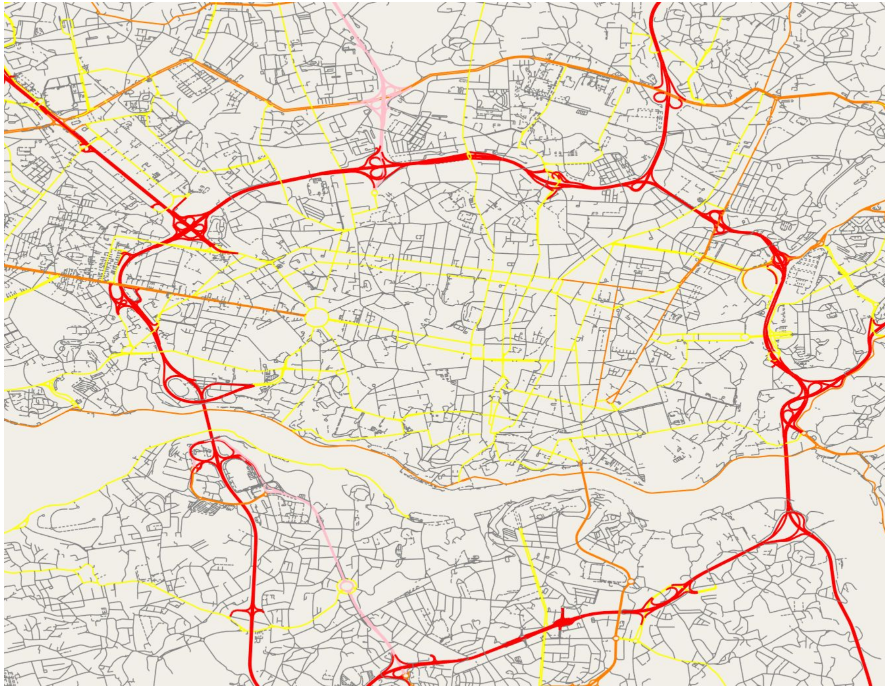
Dataset

Map

A graph representing road network in Porto Metropolitan Area (AMP)

Data extracted from OpenStreetMap

- Original file: 289.5 MB
- Filtered data:
  - `AMP.nodes`: 9.0 MB; 304,345 nodes
  - `AMP.edges`: 8.8 MB; 568,735 directed weighted edges



Dataset

Trips

A list of all 1,710,669 **taxi trips** of all 442 taxis in the city of Porto from 01/07/2013 to 30/06/2014

Data from **Kaggle** competition PKDD 15 (I)

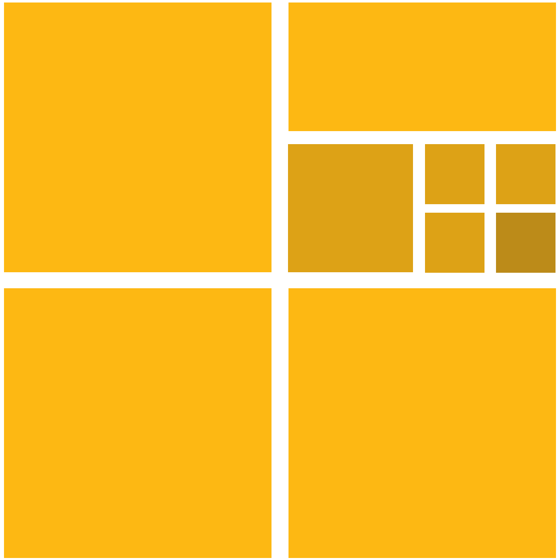
- Original file: 1.9 GB
- Filtered out runs with:
  - Missing data: 10
  - Speed errors: 173,909
  - Coordinate errors: 5,609
- Processed file: 1.5 GB
  - 1,531,135 trips
  - 72,227,758 coordinates

Using 200k trips

2-d trees

Pseudocode

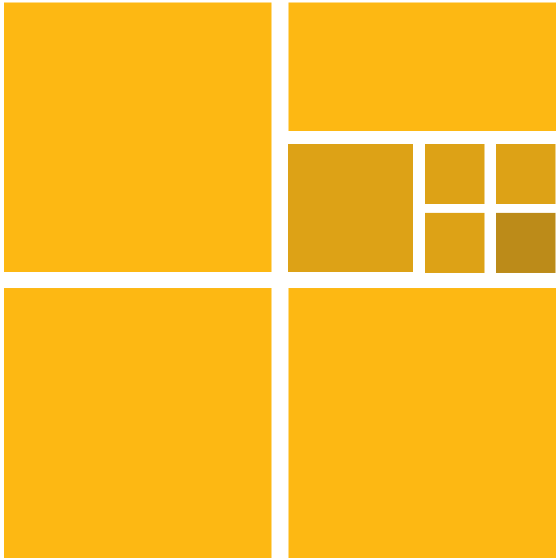
```
build(points):
    S = points.size
    N = 2^⌈log2(S)⌉ // Next power of 2
    c[N]
    c[0:S-1] = points
    c[S:] = points[S-1] // To fill all values of c
    For level = 0:log2(N)-1
        n = 2^level
        xAxisActive = (level % 2 == 0)
        For i = 0:N/n
            x = i*n
            If xAxisActive: sortByX(c[x:x+n-1])
            Else:           sortByY(c[x:x+n-1])
```



2-d trees

Pseudocode

```
dBest = INF
search(r):
    If(r is leaf):
        d = dist(r.point, p)
        If(d < dBest): dBest = d
    Else:
        v = (xAxisActive ? p.x : p.y)
        child      = (v < median ? r.lchild : r.rchild)
        otherChild = (v < median ? r.rchild : r.lchild)
        search(child)
        If(|v - r.median| < dBest):
            search(otherChild)
```



2-d trees

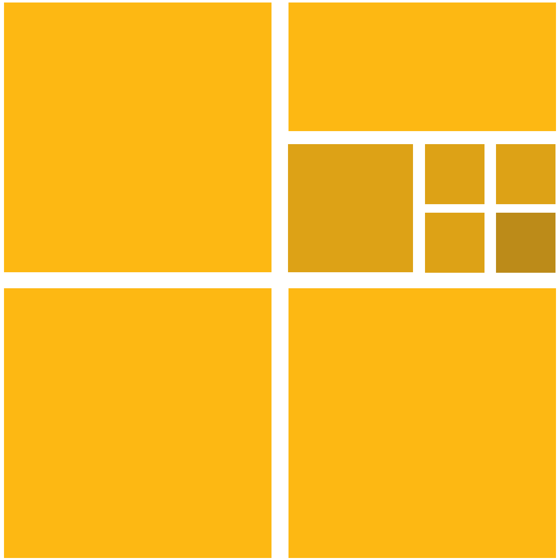
Complexity analysis

Build data structure:

- Time: $O(N \log N)$
- Space: $O(N)$

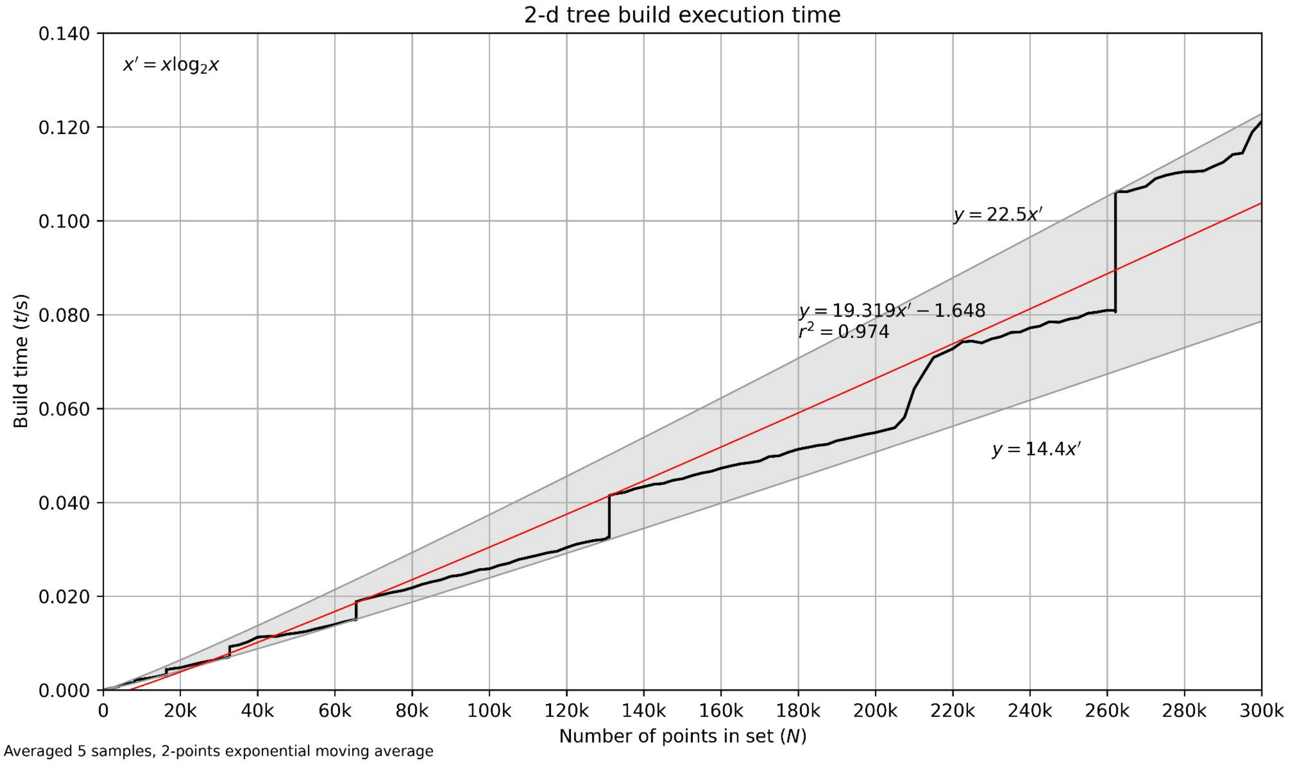
Query:

- Time: $\Theta(\log N)$, $O(N)$
- Space: $O(\log N)$



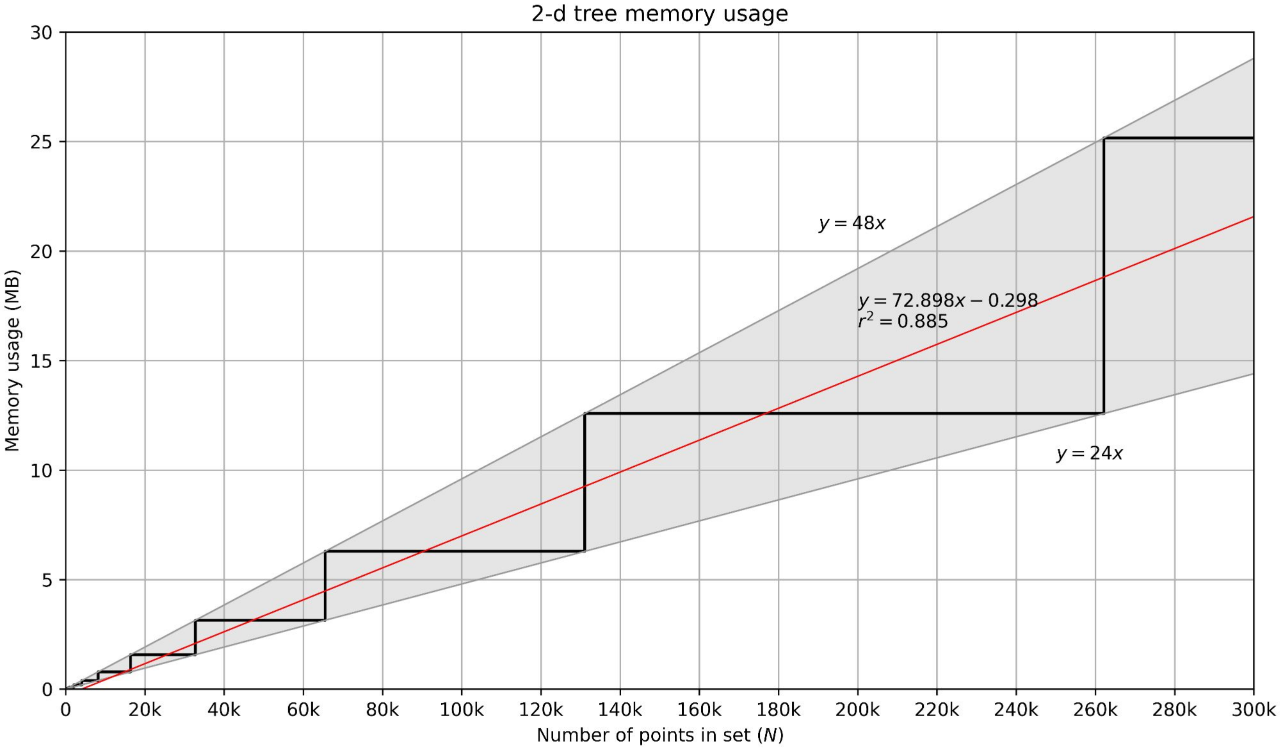
2-d trees

Empirical analysis



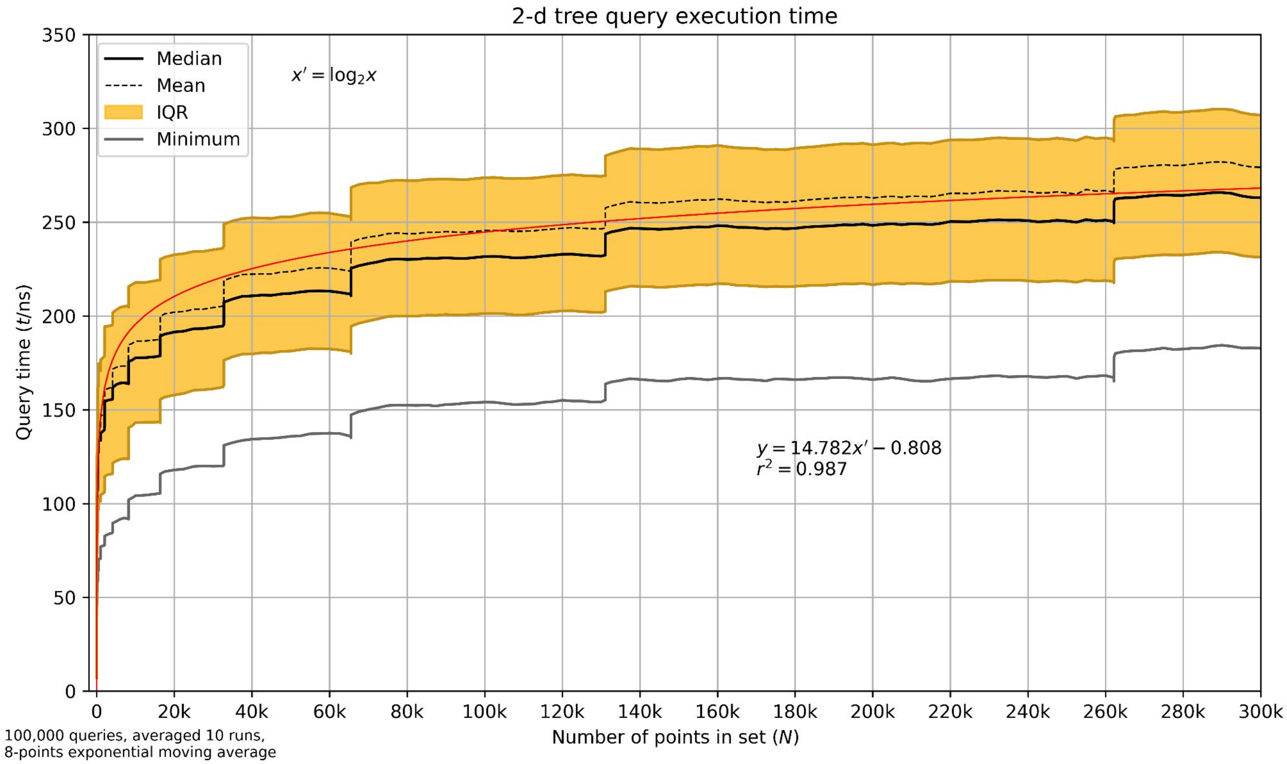
2-d trees

Empirical analysis



2-d trees

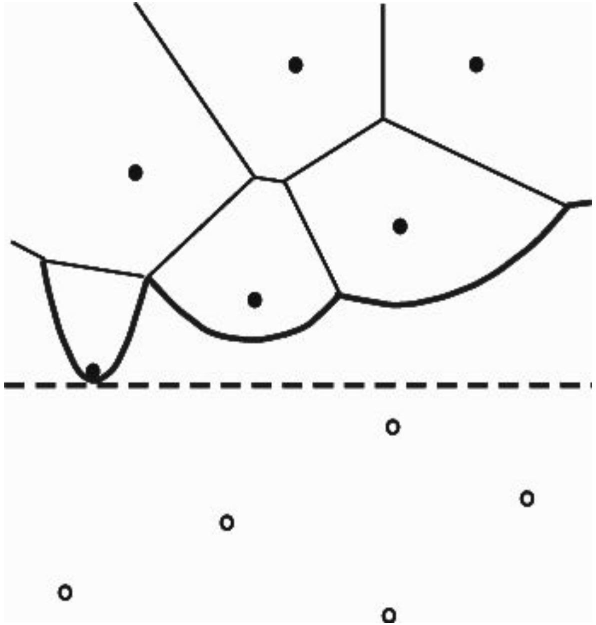
Empirical analysis



Fortune’s Algorithm

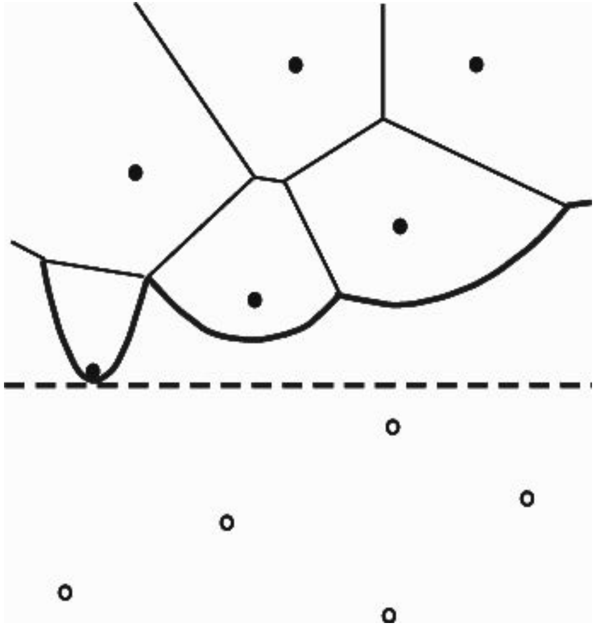
Pseudocode

```
Fill event queue with site events for each input site.
While the event queue still has items in it:
     If the next event on the queue is a site event:
          Add the new site to the beachline
     Otherwise it must be an edge-intersection event:
          Remove the squeezed cell from the beachline
Cleanup any remaining intermediate state
```



Fortune’s Algorithm

Complexity analysis

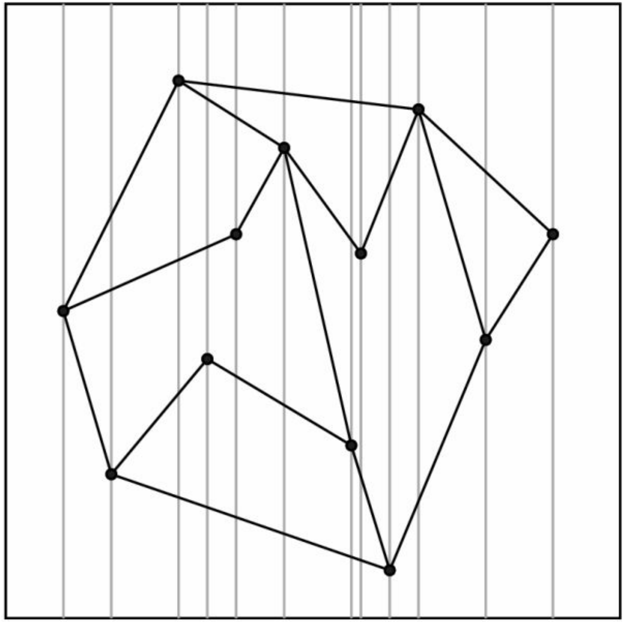


- Time: ***O(N* log *N)***
- Space: ***O(N)***

Slab decomposition

Pseudocode

```
build(edges):
    dictEvents = {} // Dictionary of lists of events
    For e ∈ edges:
        xl, xr = e.start.x, e.end.x
        dictEvents[xl].push({true , e})
        dictEvents[xr].push({false, e})
    slabs = {} // Dictionary of sets
    prevSlab, curSlab = {} // Sets of edges, sorted by Y
    For (xl, events) ∈  dictEvents:
        curSlab = prevSlab
        For (b, e) ∈ events: If(!b) curSlab.remove(e)
        For (b, e) ∈ events: If( b) curSlab.insert(e)
        slabs[xl] = curSlab
        prevSlab = curSlab
```

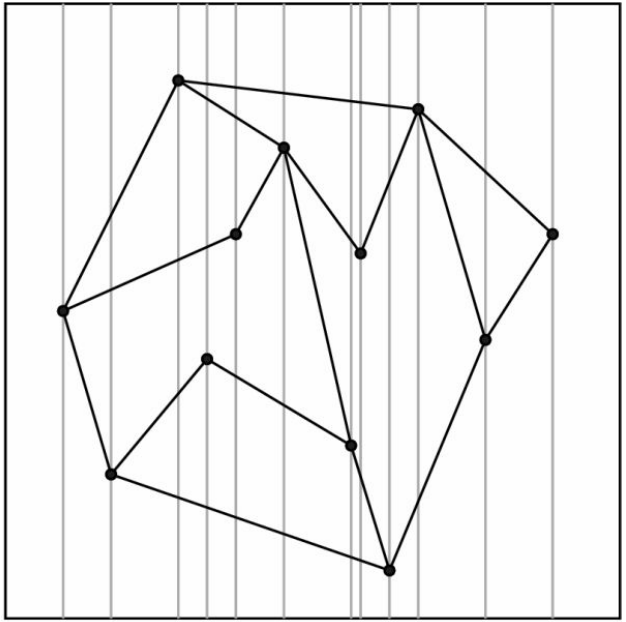


Slab decomposition

Pseudocode

```
query(p):
    slab = slabs.lowerBound(p.x)-1
    edge = slab.lowerBound(p.y)
    Return edge.siteBelow
```

A slab can be a set or a sorted vector, as long as it provides binary search



Slab decomposition

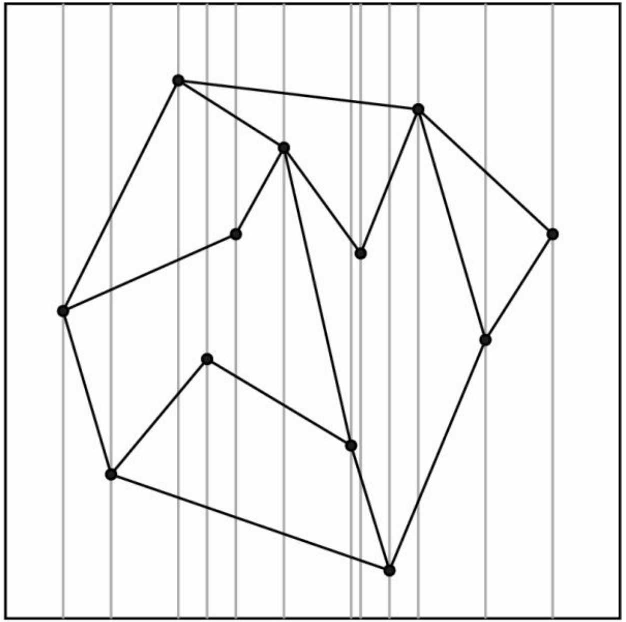
Complexity analysis

Build data structure:

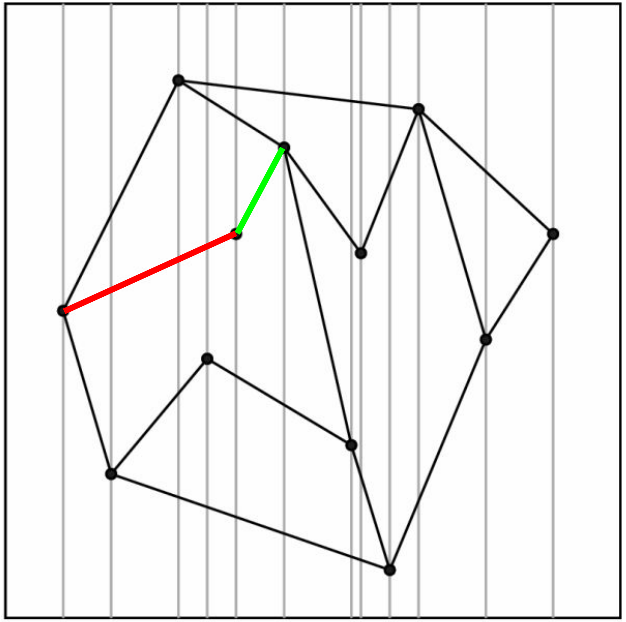
- Time: $\boldsymbol{O(N^2 \log N)}$
- Space: $\boldsymbol{O(N^2)}$

Query:

- Time: $\boldsymbol{O(\log N)}$
- Space: $\boldsymbol{O(1)}$



But $\boldsymbol{O(N^2)}$ space is too much, because N = 300k.
This is `(300k)² × sizeof(Edge*) = 720 GB`



What is the difference
between these two slabs?

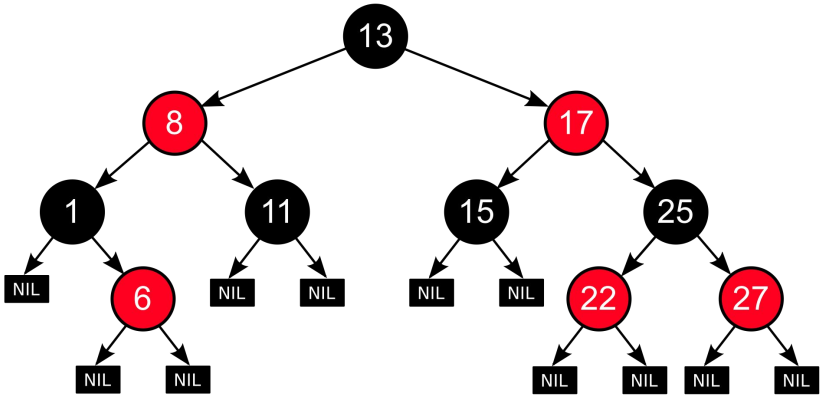
We only remove the **red** edge and add the **green** edge.

But we are copying the whole left slab, just to remove one edge and add one edge!

If we could somehow preserve past versions in a memory-efficient way...

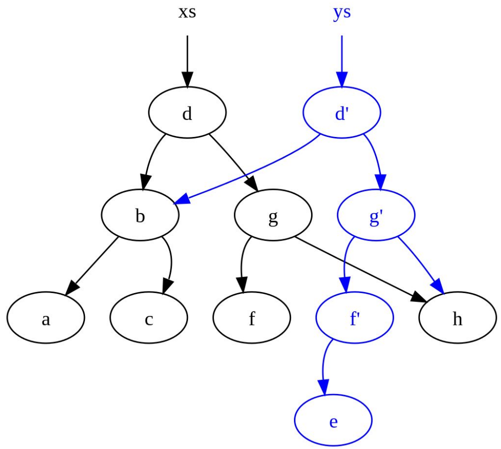
Slab decomposition

Red-black tree



A specific implementation of a self-balancing binary tree

Persistent tree



Memory-efficient:old tree is unchanged, new tree reuses as many nodes as possible

Slab decomposition

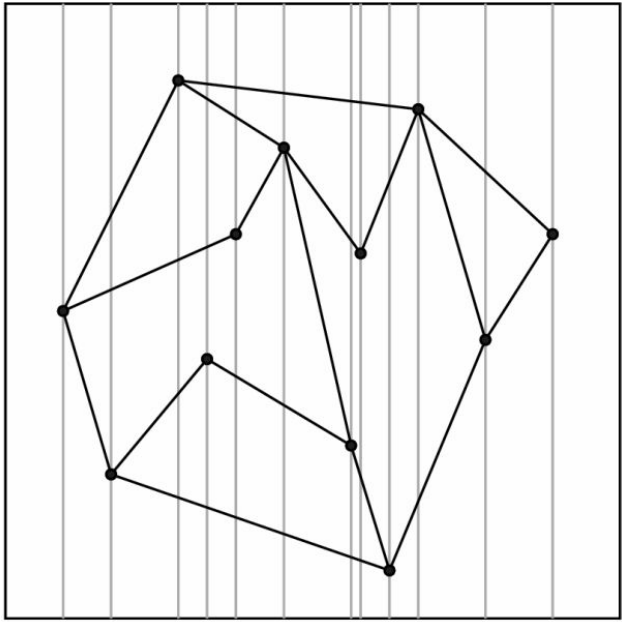
Complexity analysis (w/ persistent RB trees)

Build data structure:

- Time: ***O*(*N* log *N*)**
- Space: ***O*(*N*)**

Query:

- Time: ***O*(log *N*)**
- Space: ***O*(1)**



VStripes

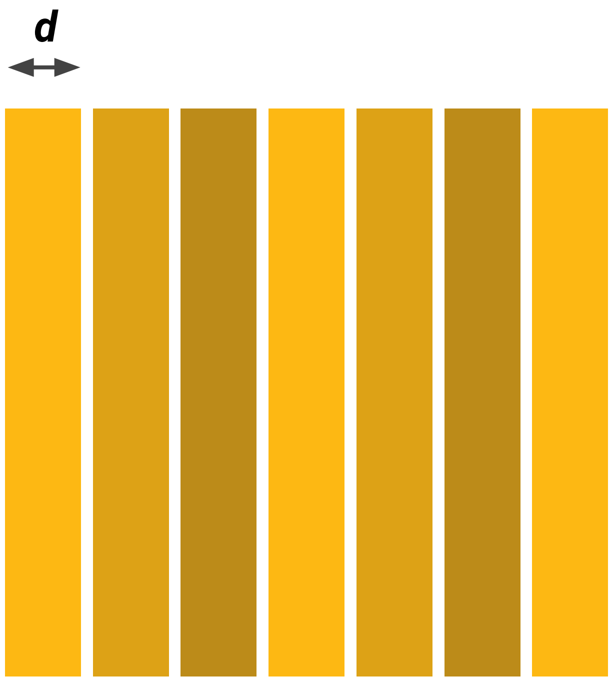
Assume upper bound of distance to solution is *d*. So we only need to search in a radius of *d*! Parametrized with *d*

Build:

1. Split space into vertical stripes of width *d*
2. In each stripe, sort points by y-coordinate

Search of point *(x, y)*:

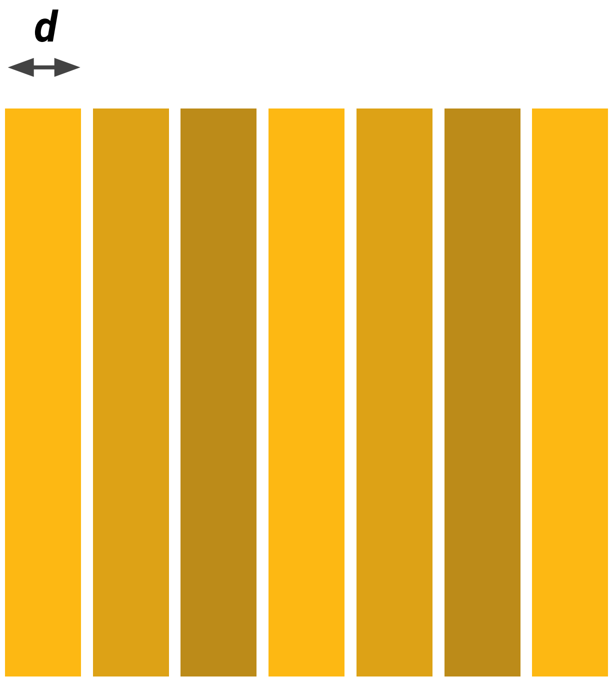
1. Find the stripe *i* where the point would be
2. Binary search to find candidate points *(x',y')* complying with $y-d \leq y' \leq y+d$
3. Iterate over all candidates
4. Repeat for stripes *i-1* and *i+1*



VStripes

Pseudocode

```
build(d, points):
    stripes = []
    sortByX(points)
    xMin, xMax = points[0].x, points[end].x
    l, r = xMin, xMin+d
    i = 0
    While l <= xMax:
        stripe = []
        While i < |points| && points[i].x < r:
            stripe.append(points[i++])
        sortByY(stripe)
        stripes.append(stripe)
        l = r
        r += d
```

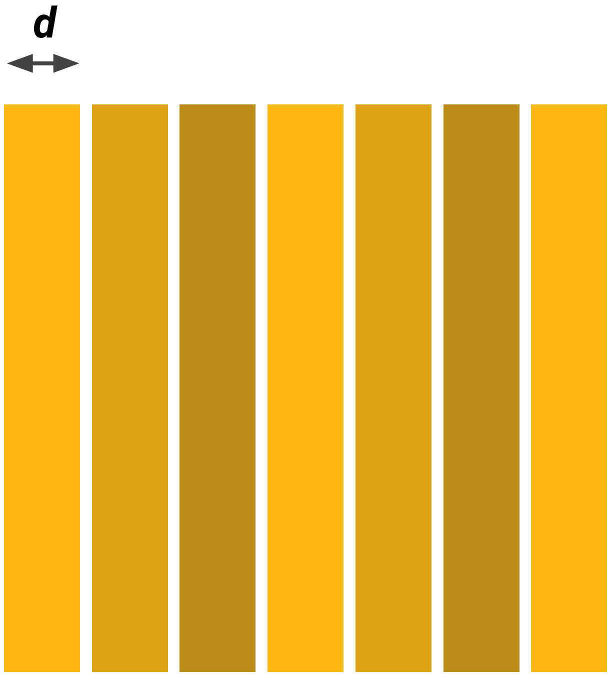


VStripes

Pseudocode

```
cBest
dBest = INF
search(p):
    i = ⌊(p.x-xMin)/d⌋
    i = min(|stripes|-1, max(0, i))
    checkStripe(p, i)
    If i-1 >= 0       : checkStripe(p, i-1)
    If i+1 < |stripes|: checkStripe(p, i+1)
    return cBest

checkStripe(p, i):
    stripe = stripes[i]
    l = stripe.lowerBoundByY(p.x-d)
    While l < |stripe| && stripe[l] < p.x+d:
        c = stripe[l++]
        d = dist(c, p)
        If d < dBest: cBest, dBest = c, d
```



VStripes

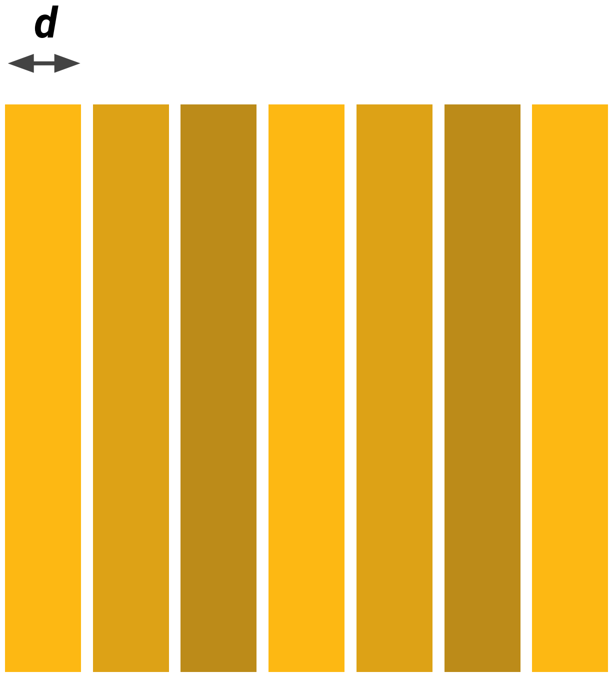
Complexity analysis

Build data structure:

- Time: $O(N \log N)$
- Space: $O(N)$

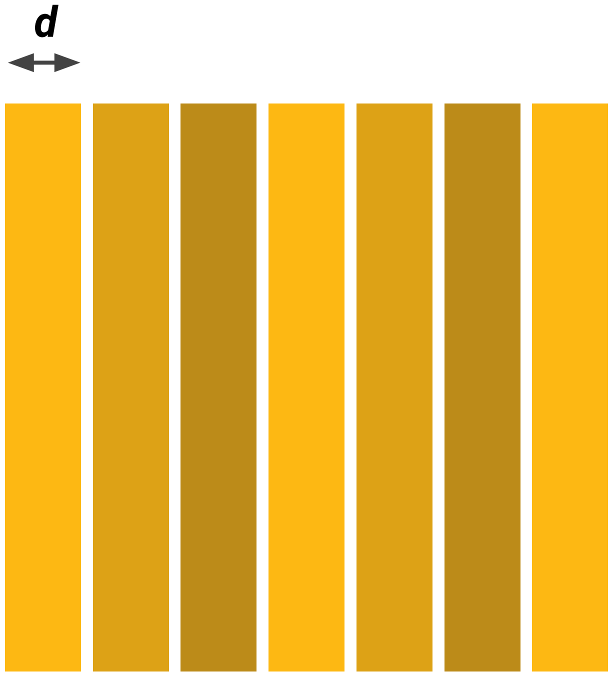
Query:

- Time: $\Theta(\log N)$, $O(N)$
- Space: $O(1)$



The problem with VStripes

- What if *d* is too small?
  - No solution is found!
- What if *d* is too large?
  - Degenerates into brute-force approach
- What if there are different optimal values for *d* depending on the region of the plane?
  - We get worst-case performance more commonly



DeepVStripes

- Instead of one VStripes data structure, we have many, with different values of *d*
- If we can't find a solution assuming *d*, we try to assume *2d*, or *4d*, or *8d*, …
- Parametrized with:
  - *d*, estimate of the mean distance to the best solution in lowest level
  - *L*, the number of levels

- If x amplitude is *X*, we can assure correctness by setting $L = \lceil \log_2(X/d) \rceil + 1$: in worst case, goes to highest VStripes, where there's only 1 stripe, and degenerates into brute-force

DeepVStripes

Pseudocode

```
build(d, L, points):
    vstripesVtr = []
    For i = 0:L
        vstripesVtr.append(VStripes.build(d, points))
        d *= 2

search(points):
    i = 0
    Do:
        vstripesVtr[i++].search(points)
    While i < L && dBest == INF
```

DeepVStripes

Complexity analysis

Build data structure:

- Time: $\boldsymbol{O(L\ N \log N)}$
- Space: $\boldsymbol{O(L\ N)}$

Query:

- Time: $\boldsymbol{\Theta(\log N)}$, $\boldsymbol{O(N)}$
- Space: $\boldsymbol{O(1)}$

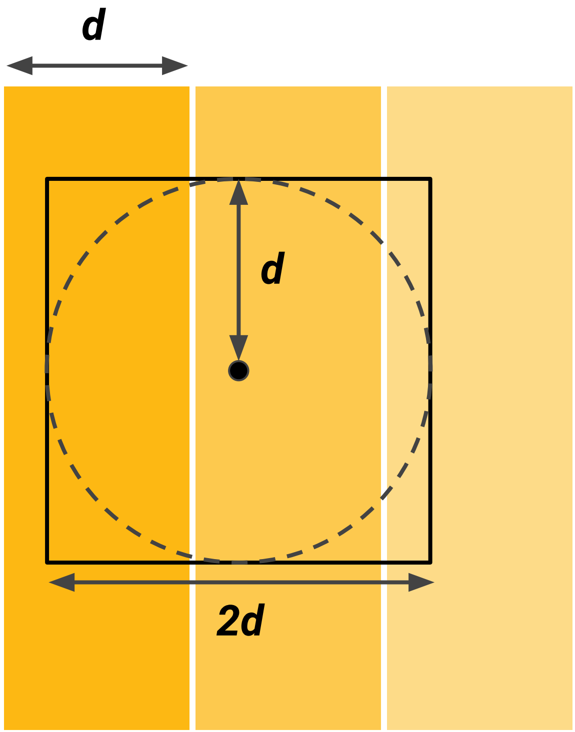
The query worst-case is rare, because we can tune *d*

DeepVStripes

Math

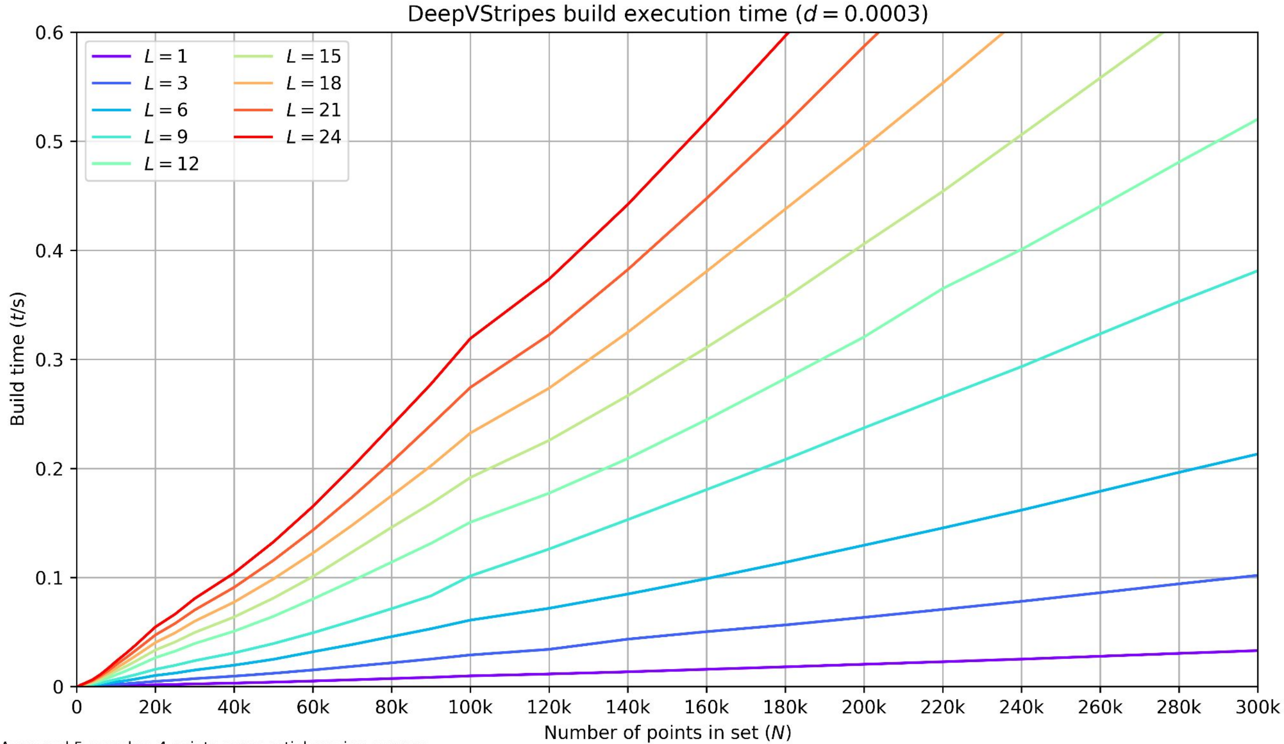
Amplitude in x and y are X and Y
Assuming uniform distribution of N points in area XY, to get 1 point in area $4D^2$ we can set $4d^2=XY/N$

Using $X = 0.3145$, $Y = 0.3742$, $N = 304345$, we get $D = 0.00031092$, which is the optimal value of d



DeepVStripes

Empirical analysis

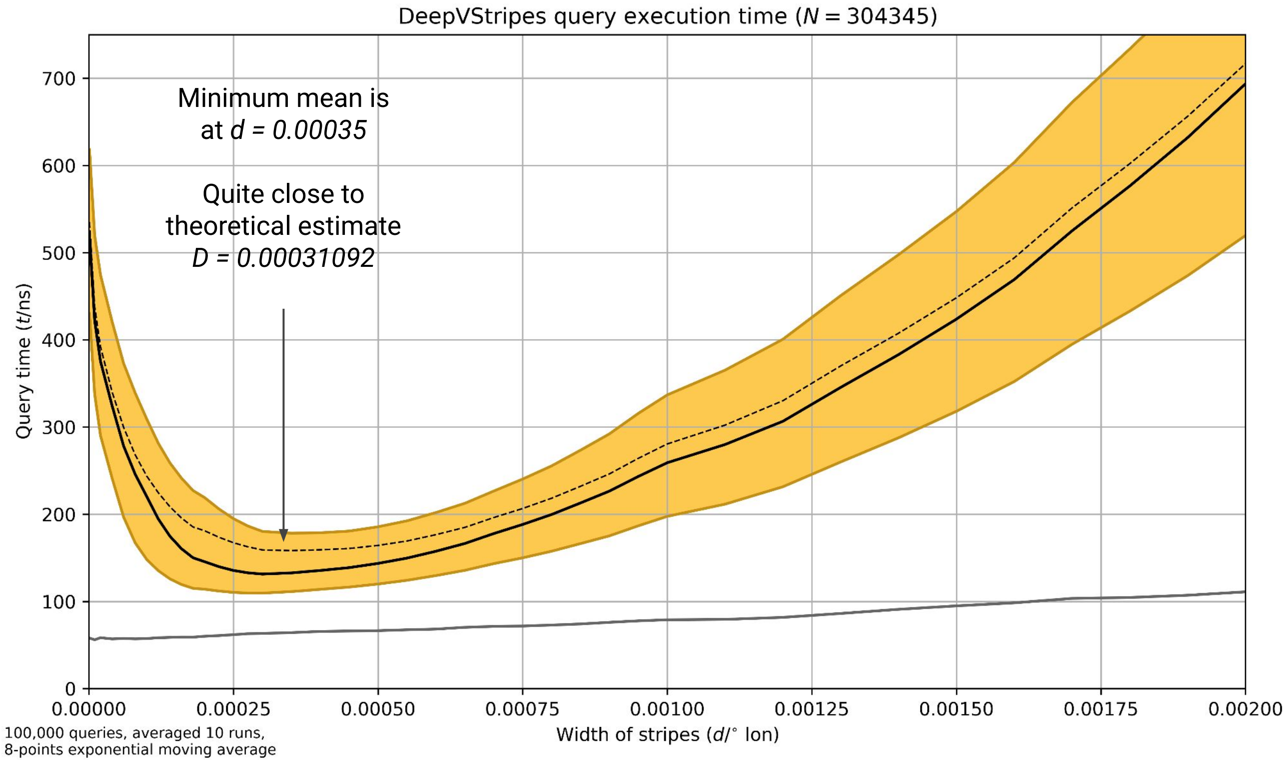


Averaged 5 samples, 4-points exponential moving average

DeepVStripes

Empirical analysis

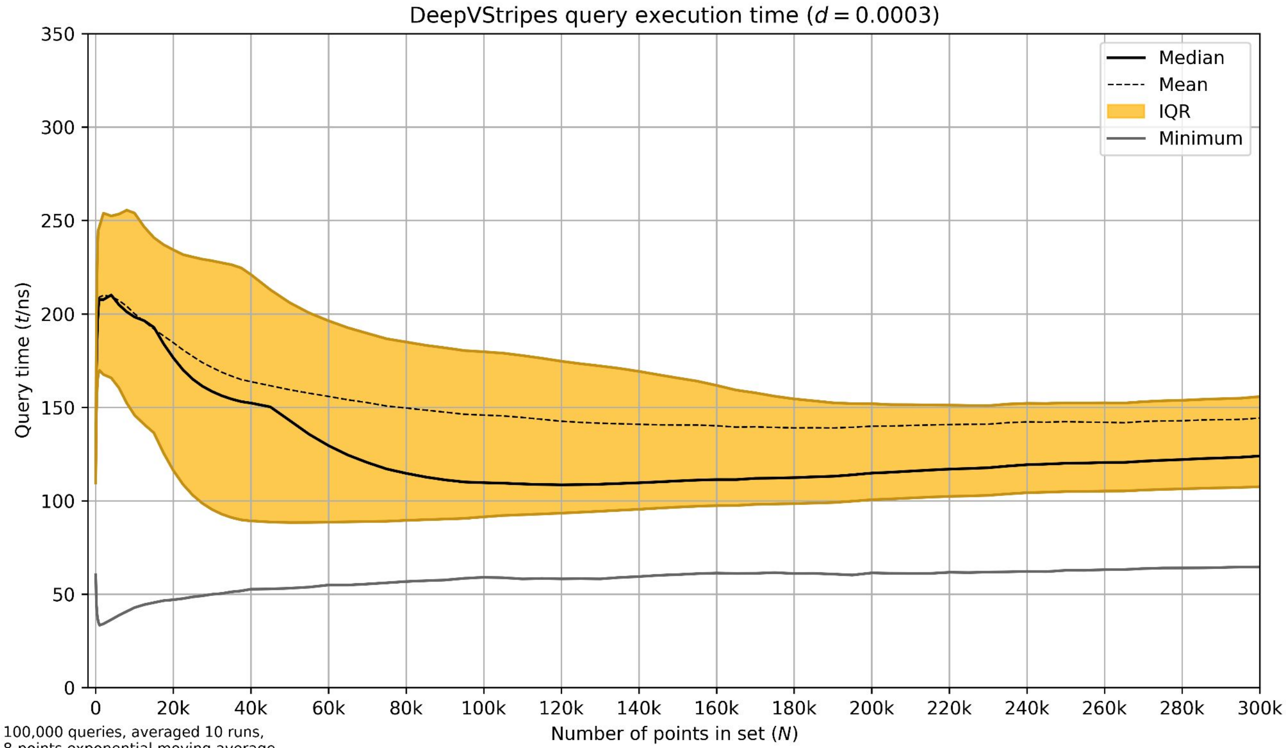
Here, climbing up to correct *d* is dominant: ***O*(log(*D*/*d*))**



Here, number of points in *4d²* is dominant: ***O*(*d*²)**

DeepVStripes

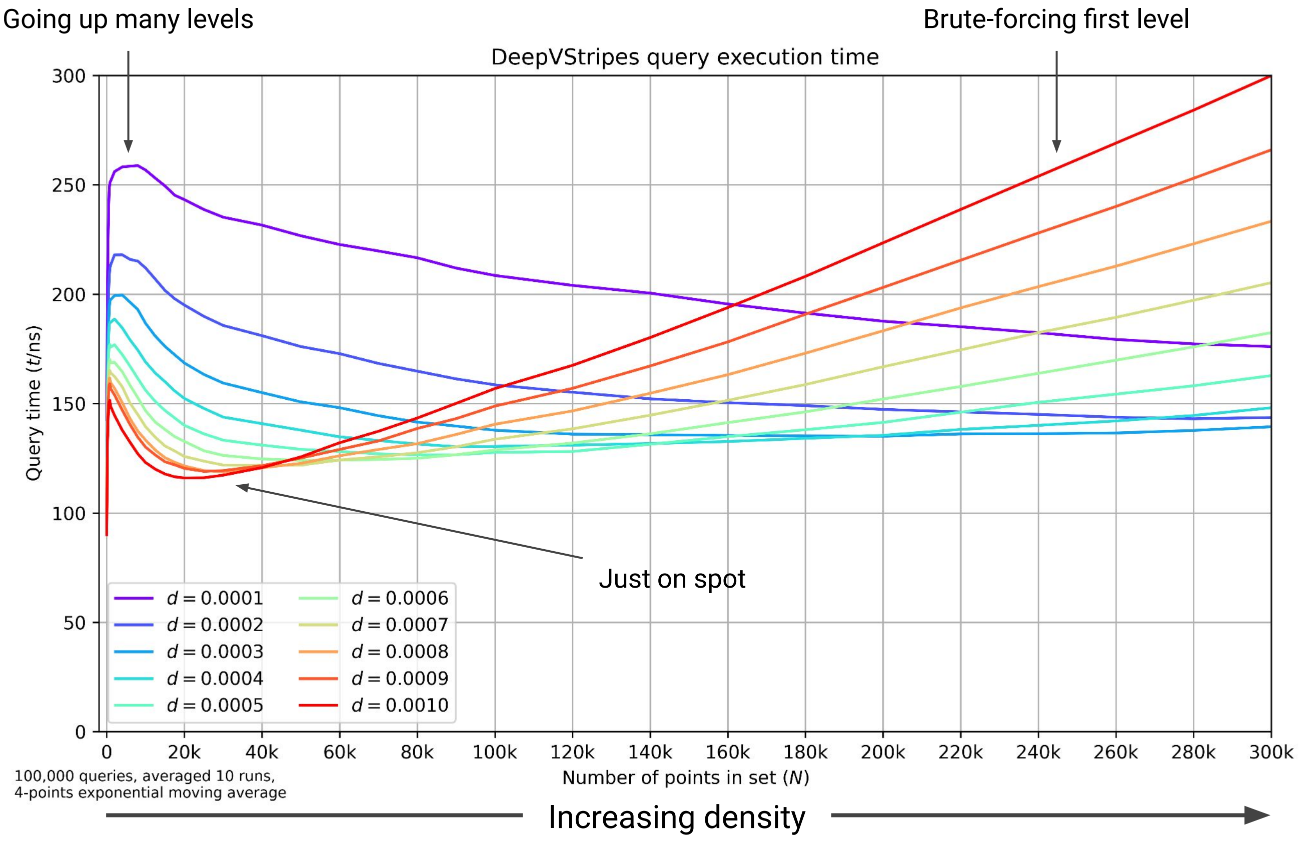
Empirical analysis



100,000 queries, averaged 10 runs,
8-points exponential moving average

DeepVStripes

Empirical analysis



Pseudocode

Fortune's algorithm

```
for each site,
    create site event e,
    e.point = current site, insert
    e into queue
while queue not empty,
    e = first event from queue
    if e is site event:
        addParabola(e.point)
    else:
        removeParabola(e.parabola)
```

```
addParabola(point p):
    if arc under p has circle event,
        remove from queue
    create arcs a1,a2,a3
    a1.site = p, a2.site = a3.site =
    site of the arc under p
    edges xl, xr = normals to a2 and a1
sites, and to a1 and a3, respectively
    replace arc under p by a2, xl, a1,
xr, a3
    check circle events for a2 and a3
```

Pseudocode

Fortune's algorithm (cont.)

```
removeParabola(parabola p)
    l,r → arc left and right of p
    if either have circle events,
remove from queue
    replace xl, p, xr by new edge x
that starts at circumcenter of l,p
and r sites.
    check circle events for l and r
```

```
check circle events (parabola p)
    if arc on left and right of p
       exist and left != right and
      if the edges by the parabola
      (xl, xr) cross in a (middle point s)

    if a = dist(middle, parabola site with
s) not under sweep line
        create circle event e,
        e.parabola = p, e.coordValue = a
        add e to queue
```

Q&A