New directions in approximate nearest neighbors for the angular distance

Thijs Laarhoven

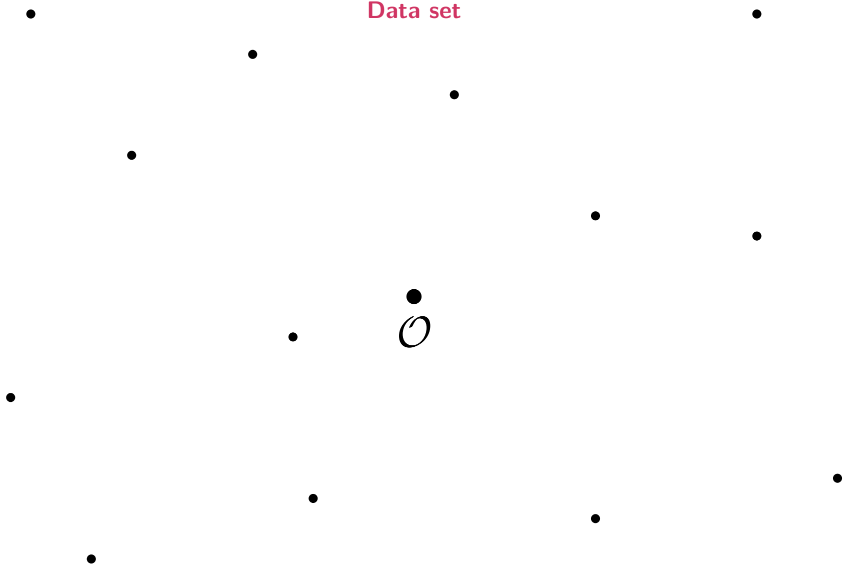
mail@thijs.com
http://www.thijs.com/

Proximity Workshop, College Park (MD), USA
(January 13, 2016)

Nearest neighbor searching

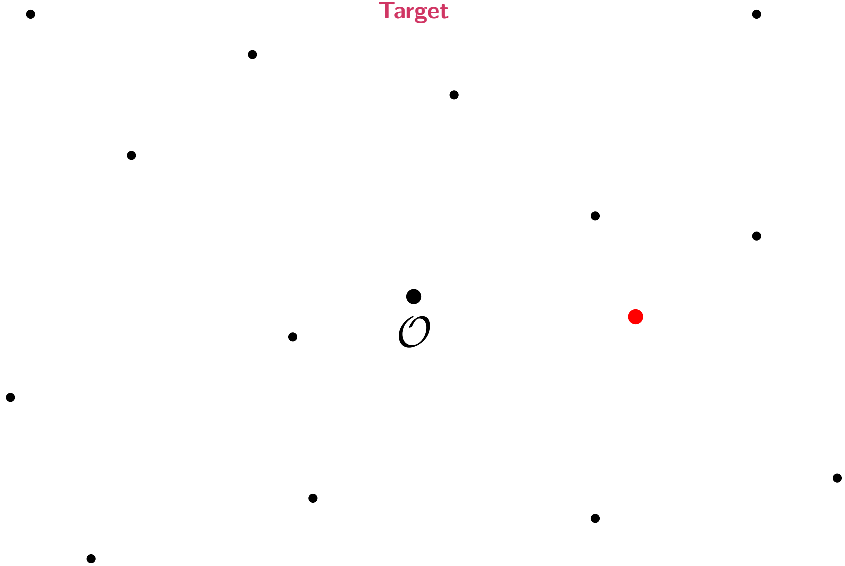
Nearest neighbor searching

Data set



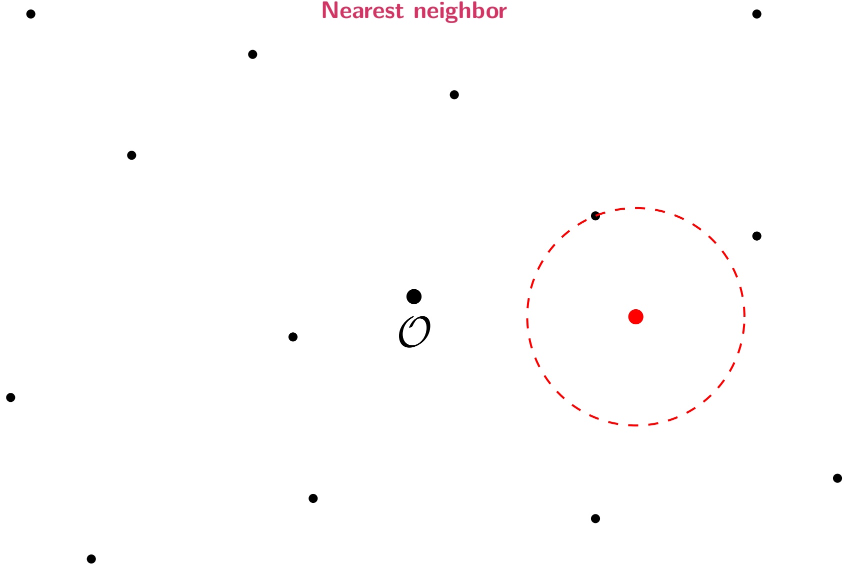
Nearest neighbor searching

Target



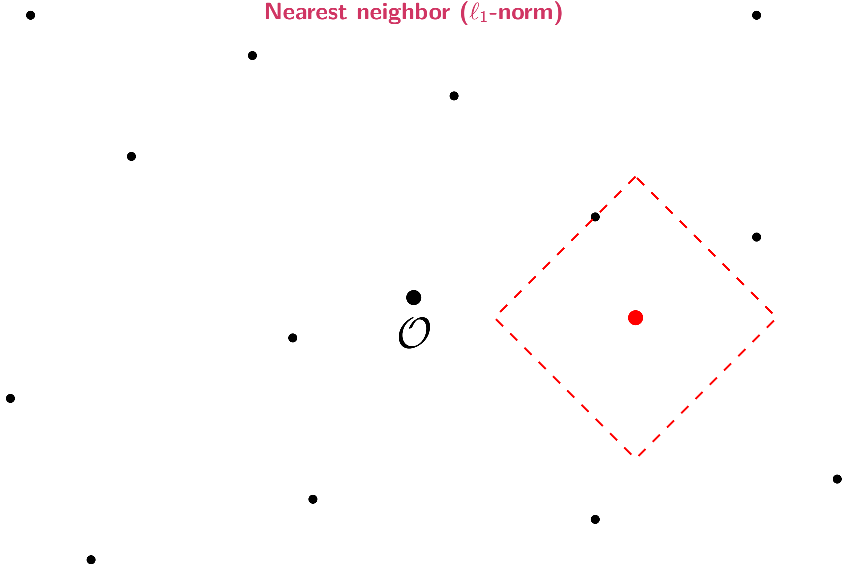
Nearest neighbor searching

Nearest neighbor



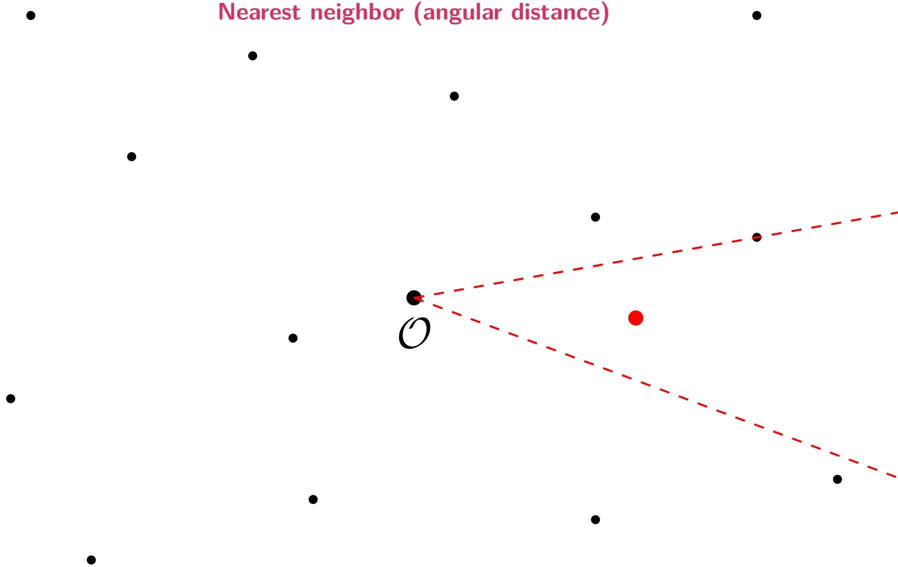
Nearest neighbor searching

Nearest neighbor (ℓ_1-norm)



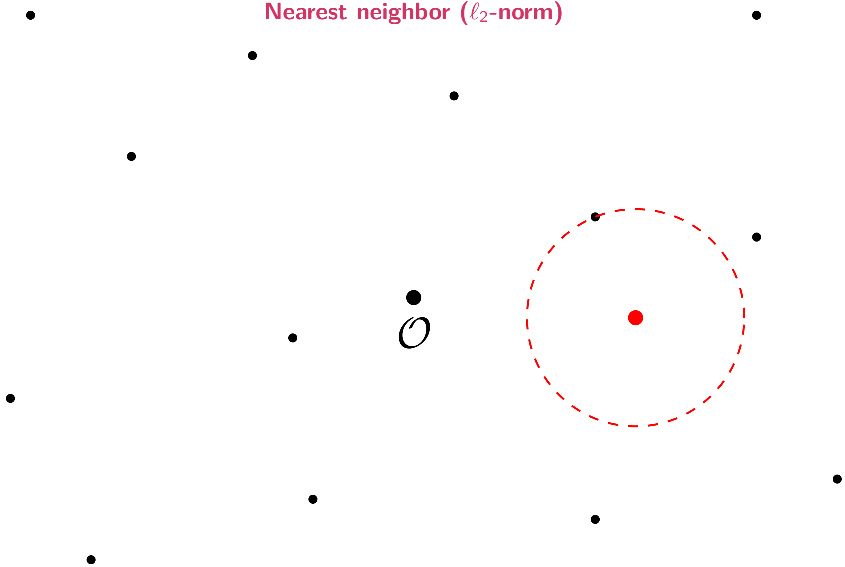
Nearest neighbor searching

Nearest neighbor (angular distance)



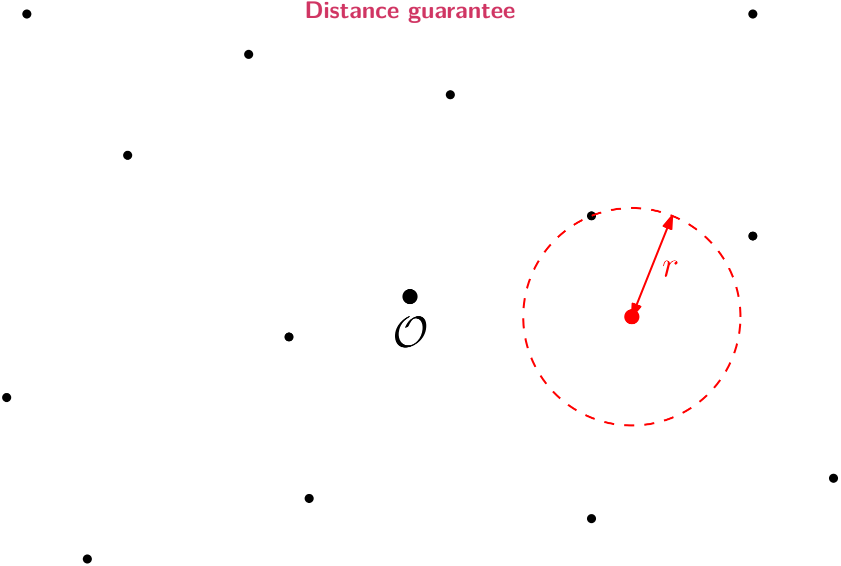
Nearest neighbor searching

Nearest neighbor (ℓ_2-norm)



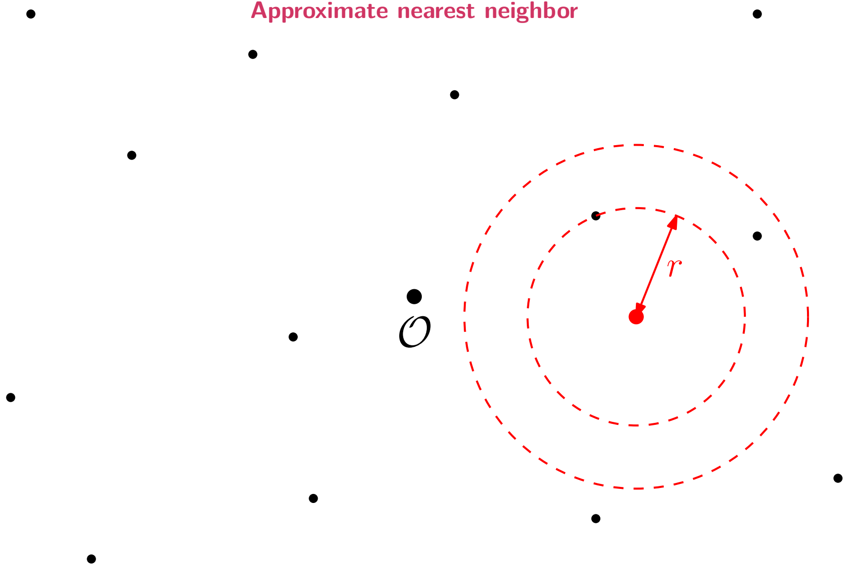
Nearest neighbor searching

Distance guarantee



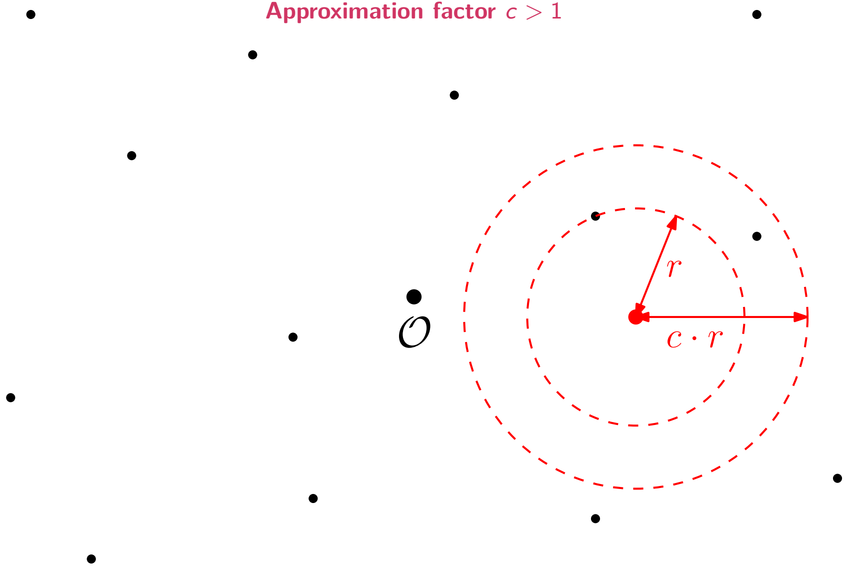
Nearest neighbor searching

Approximate nearest neighbor



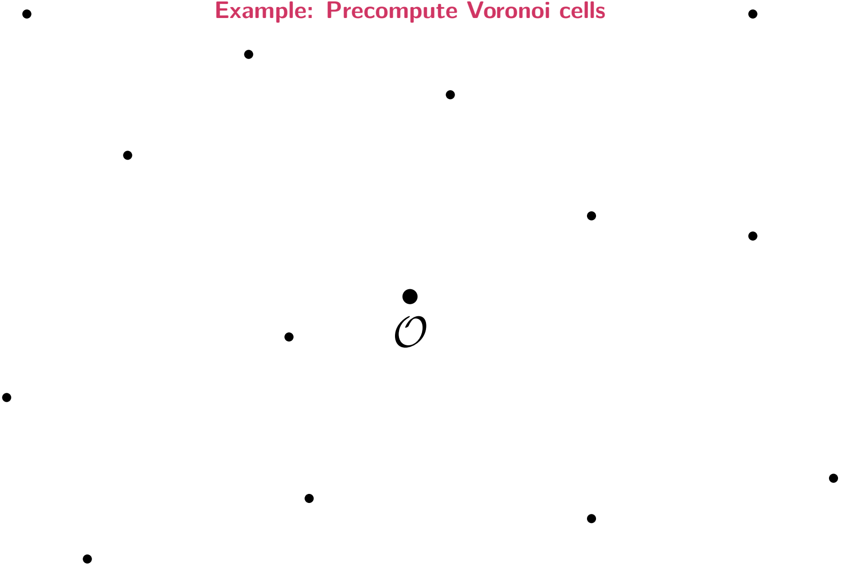
Nearest neighbor searching

Approximation factor $c > 1$



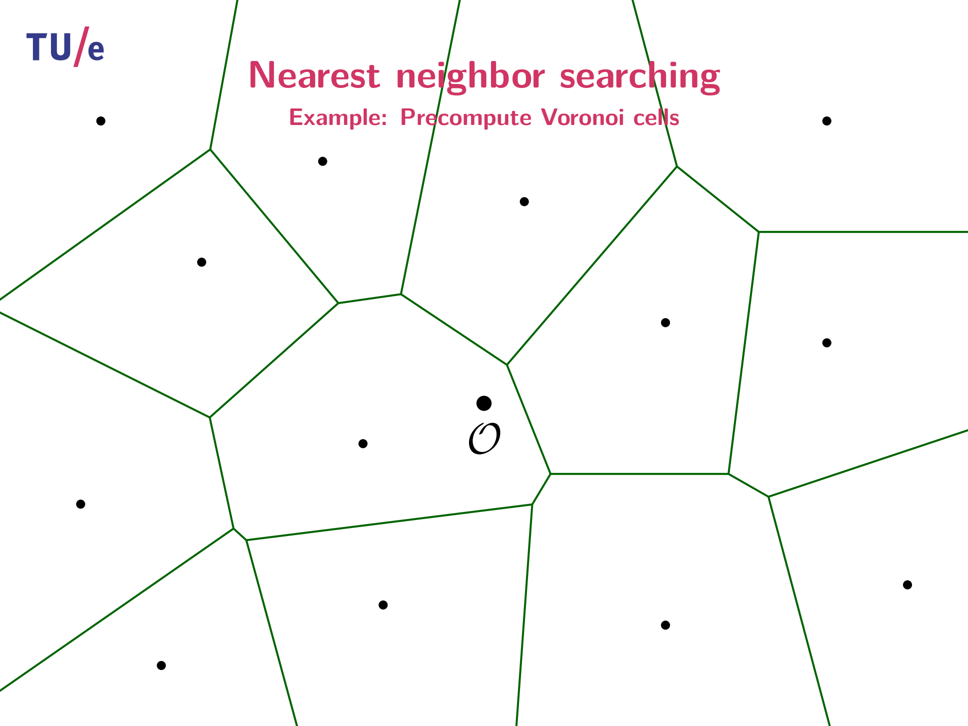
Nearest neighbor searching

Example: Precompute Voronoi cells



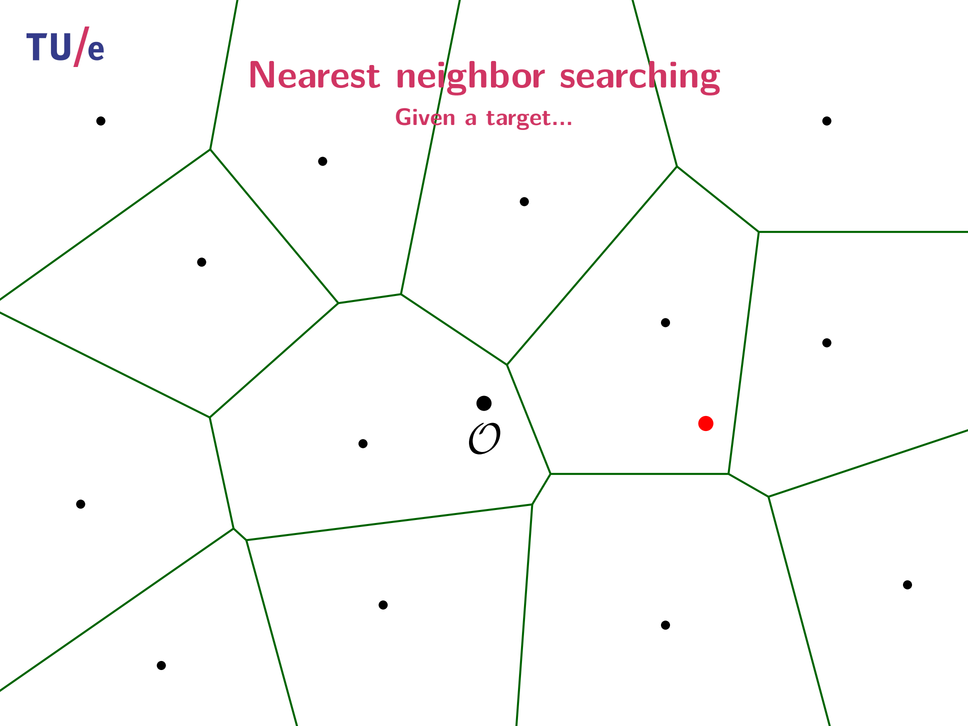
Nearest neighbor searching

Example: Precompute Voronoi cells



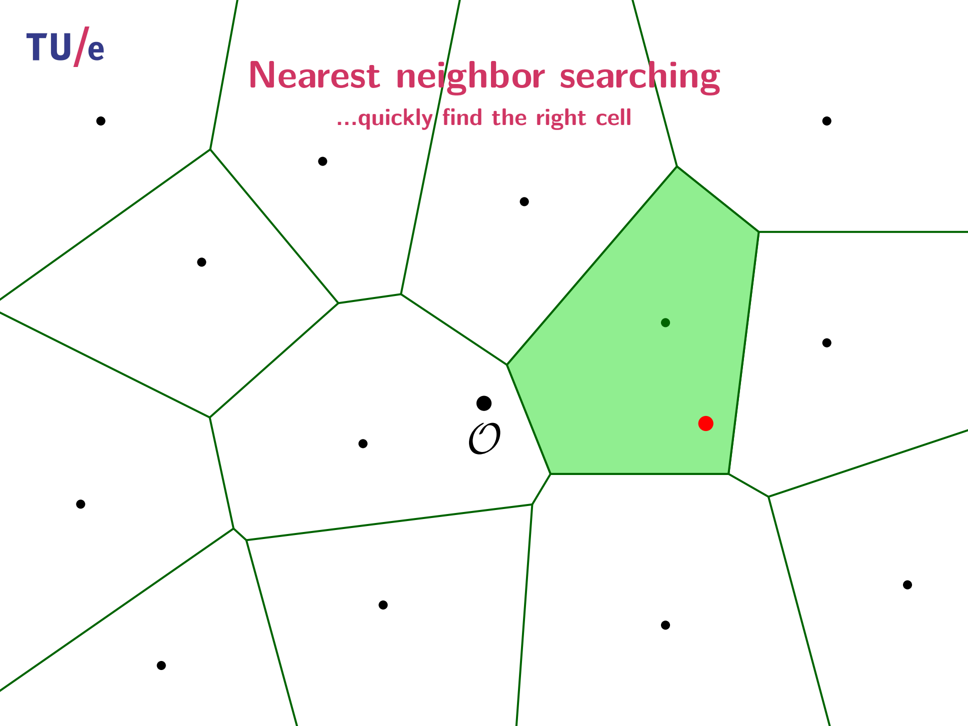
Nearest neighbor searching

Given a target...



Nearest neighbor searching

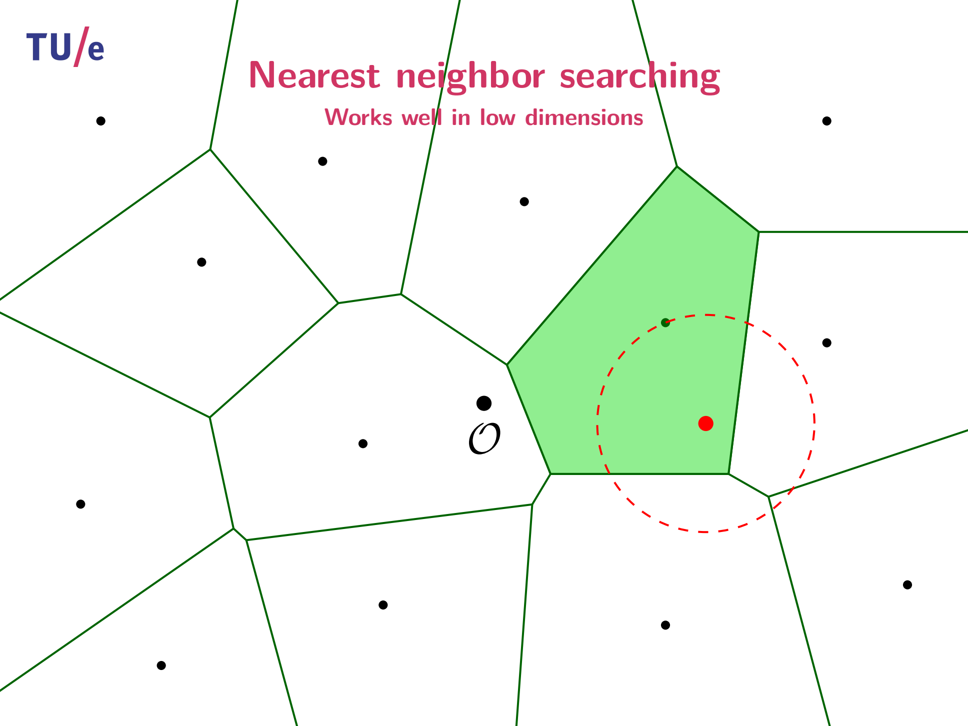
...quickly find the right cell



TU/e

Nearest neighbor searching

Works well in low dimensions



Nearest neighbor searching

Problem setting

- High dimensions d

Nearest neighbor searching

Problem setting

- High dimensions d
- Large data set of size $n = 2^{\Omega(d/\log d)}$
  - Smaller n? $\implies$ Use JLT to reduce d

Nearest neighbor searching

Problem setting

- High dimensions d
- Large data set of size $n = 2^{\Omega(d/\log d)}$
  - Smaller n? $\implies$ Use JLT to reduce d
- Assumption: Data set lies on the sphere
  - Angular NNS in $\mathbb{R}^d$ equivalent to Eucl. NNS on the sphere
  - Reduction from Eucl. NNS in $\mathbb{R}^d$ to Eucl. NNS on the sphere [AR'15]

Nearest neighbor searching

Problem setting

- High dimensions d
- Large data set of size $n = 2^{\Omega(d/\log d)}$
  - Smaller n? $\implies$ Use JLT to reduce d
- Assumption: Data set lies on the sphere
  - Angular NNS in $\mathbb{R}^d$ equivalent to Eucl. NNS on the sphere
  - Reduction from Eucl. NNS in $\mathbb{R}^d$ to Eucl. NNS on the sphere [AR'15]
- "Random" case: $c \cdot r = \sqrt{2}$
  - Random unit vectors are usually approximately orthogonal

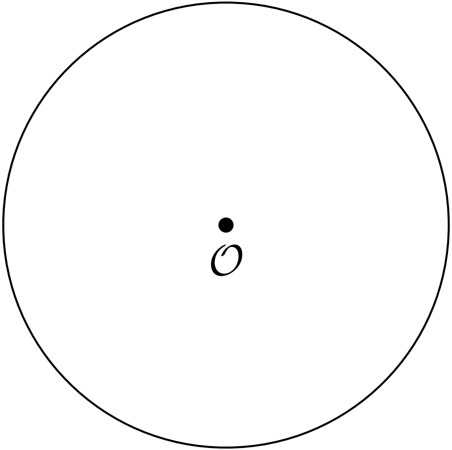
Nearest neighbor searching

Problem setting

- High dimensions d
- Large data set of size $n = 2^{\Omega(d/\log d)}$
  - Smaller n? $\implies$ Use JLT to reduce d
- Assumption: Data set lies on the sphere
  - Angular NNS in $\mathbb{R}^d$ equivalent to Eucl. NNS on the sphere
  - Reduction from Eucl. NNS in $\mathbb{R}^d$ to Eucl. NNS on the sphere [AR'15]
- "Random" case: $c \cdot r = \sqrt{2}$
  - Random unit vectors are usually approximately orthogonal
- Goal: Query time $O(n^\rho)$ with $\rho < 1$

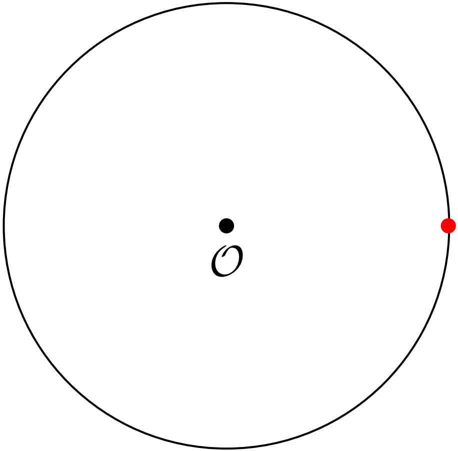
Nearest neighbor searching

“Random” instances



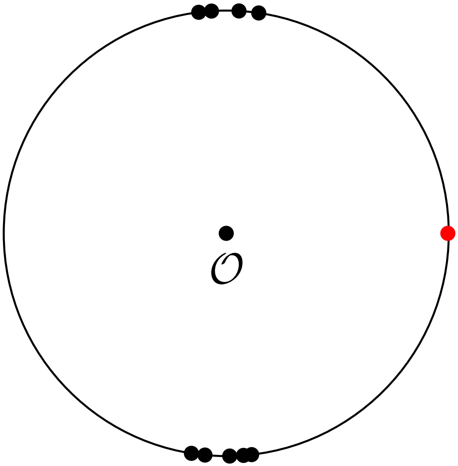
Nearest neighbor searching

“Random” instances



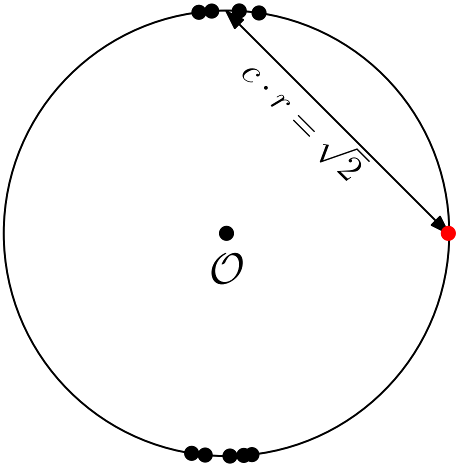
Nearest neighbor searching

“Random” instances



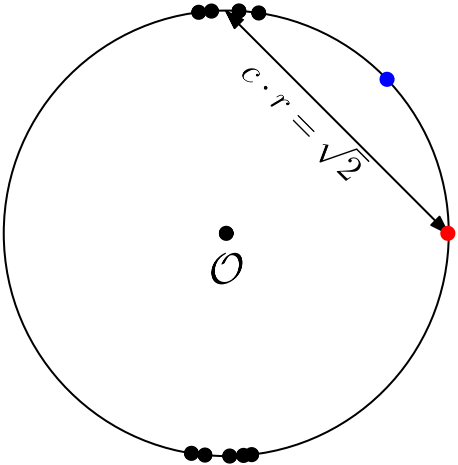
Nearest neighbor searching

“Random” instances



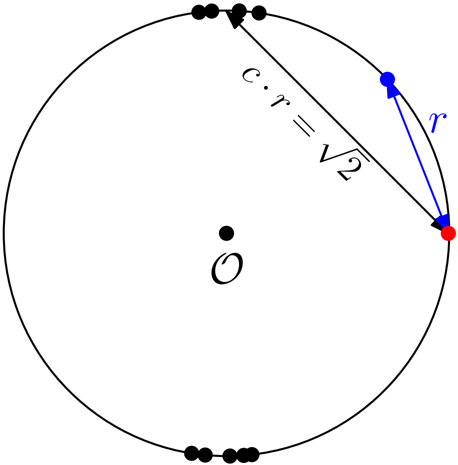
Nearest neighbor searching

“Random” instances



Nearest neighbor searching

“Random” instances



Locality-sensitive hashing

Overview

Locality-sensitive hashing

Overview

- Idea: Use nice partitions of the space
  - Nearby vectors are often in the same region
  - Distant vectors are unlikely to be in the same region

Locality-sensitive hashing

Overview

- Idea: Use nice partitions of the space
  - Nearby vectors are often in the same region
  - Distant vectors are unlikely to be in the same region
- Precomputation: Store hash tables of vectors per region
  - For each region, store contained vectors from data set
  - Rerandomization: Many partitions to increase success probability

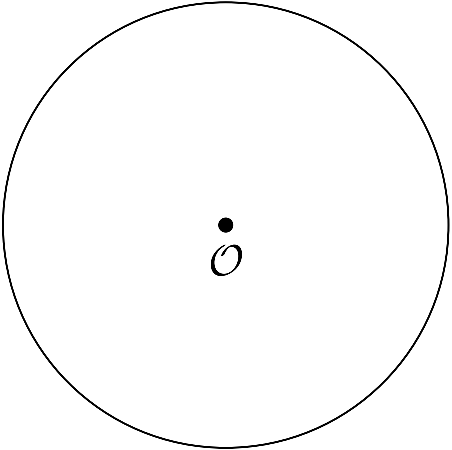
Locality-sensitive hashing

Overview

- Idea: Use nice partitions of the space
  - Nearby vectors are often in the same region
  - Distant vectors are unlikely to be in the same region
- Precomputation: Store hash tables of vectors per region
  - For each region, store contained vectors from data set
  - Rerandomization: Many partitions to increase success probability
- Query: Check hash tables for collisions
  - Compute target's region for each hash table
  - Check corresponding buckets for potential nearest neighbors
  - Reduces search space before doing a linear search

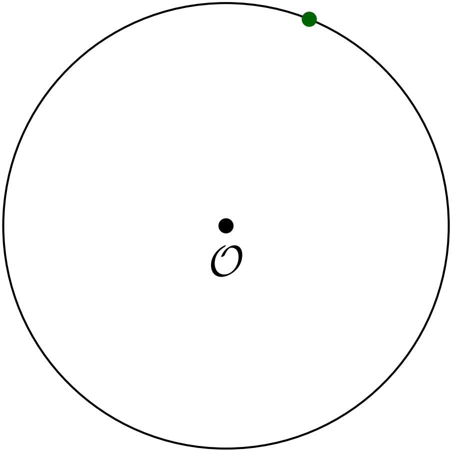
Hyperplane LSH

[Charikar, STOC'02]



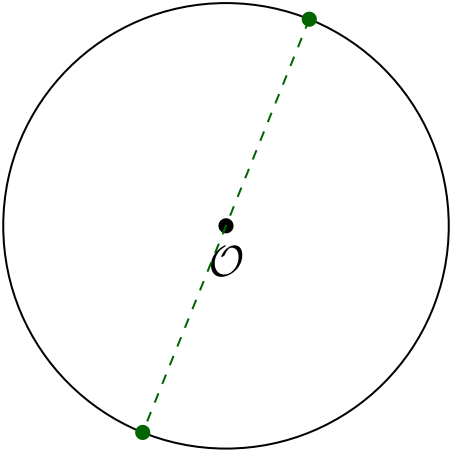
Hyperplane LSH

Random point



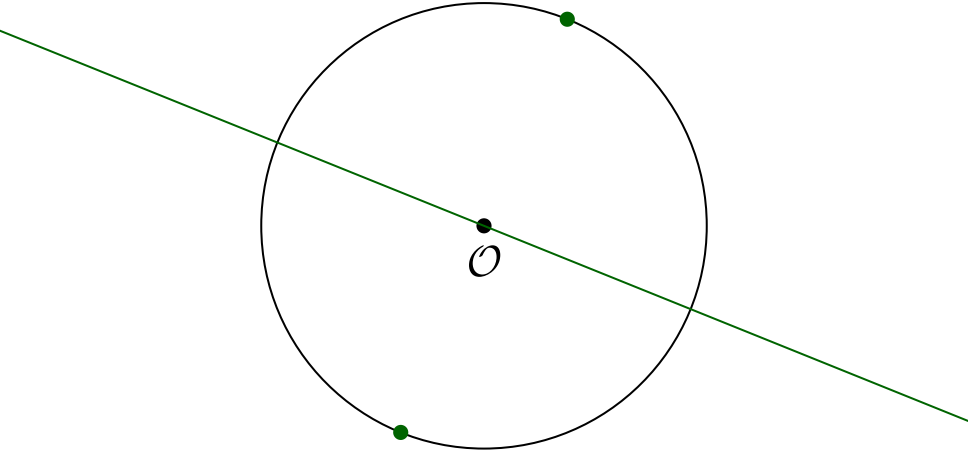
Hyperplane LSH

Opposite point



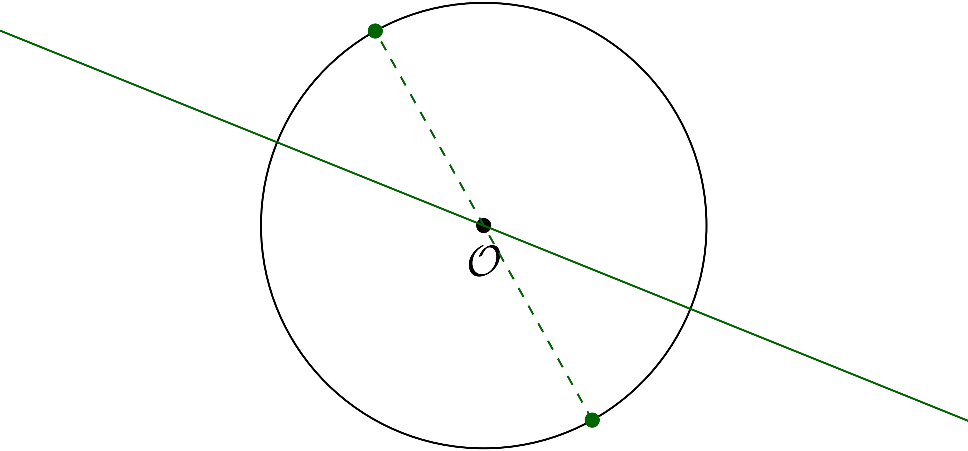
Hyperplane LSH

Two Voronoi cells



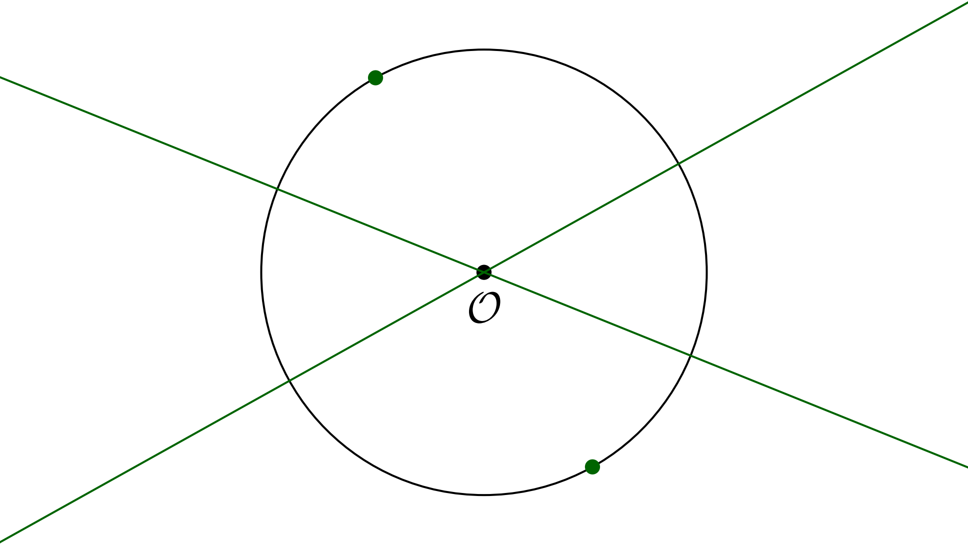
Hyperplane LSH

Another pair of points



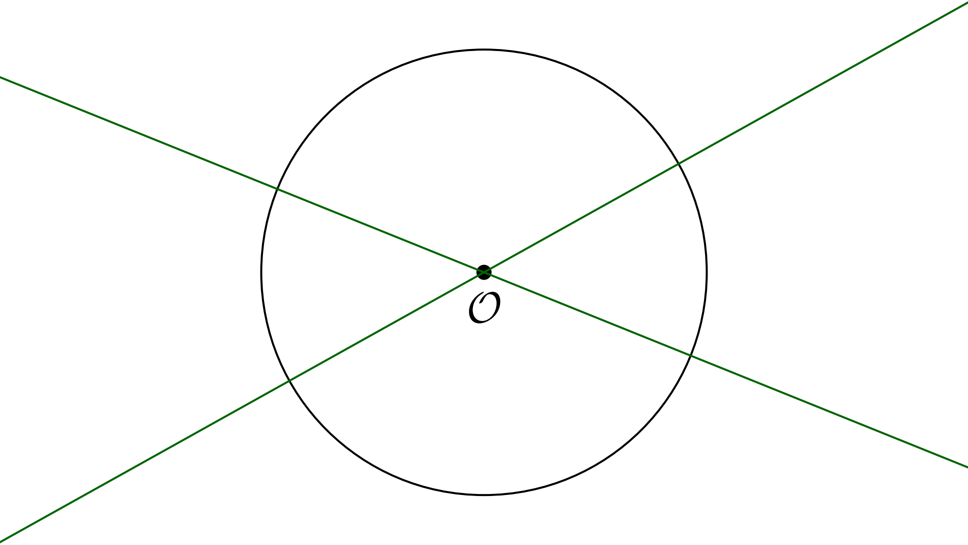
Hyperplane LSH

Another hyperplane



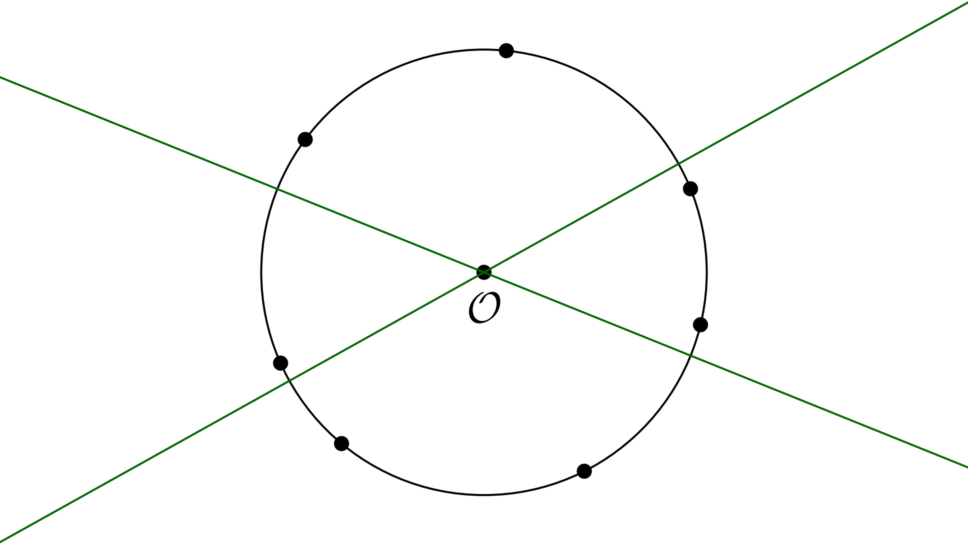
Hyperplane LSH

Defines partition



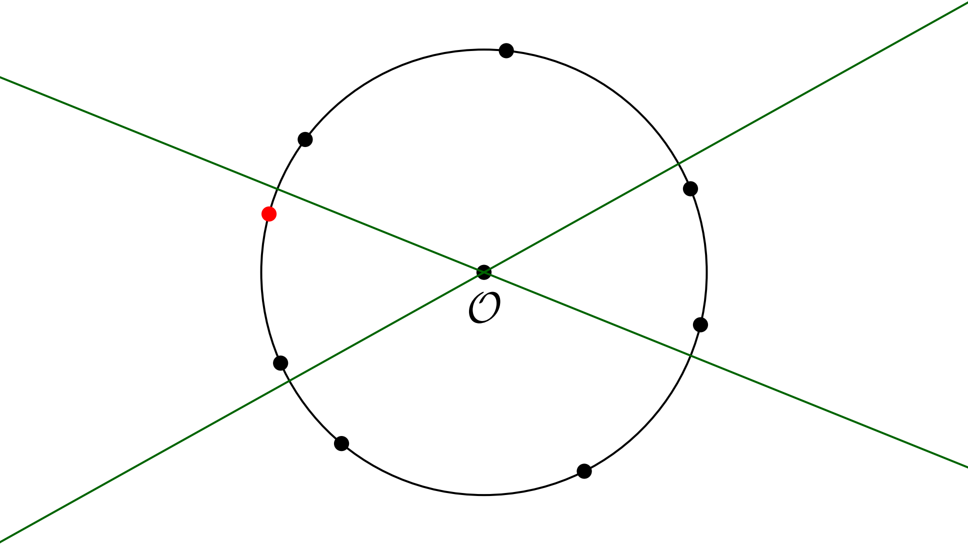
Hyperplane LSH

Preprocessing



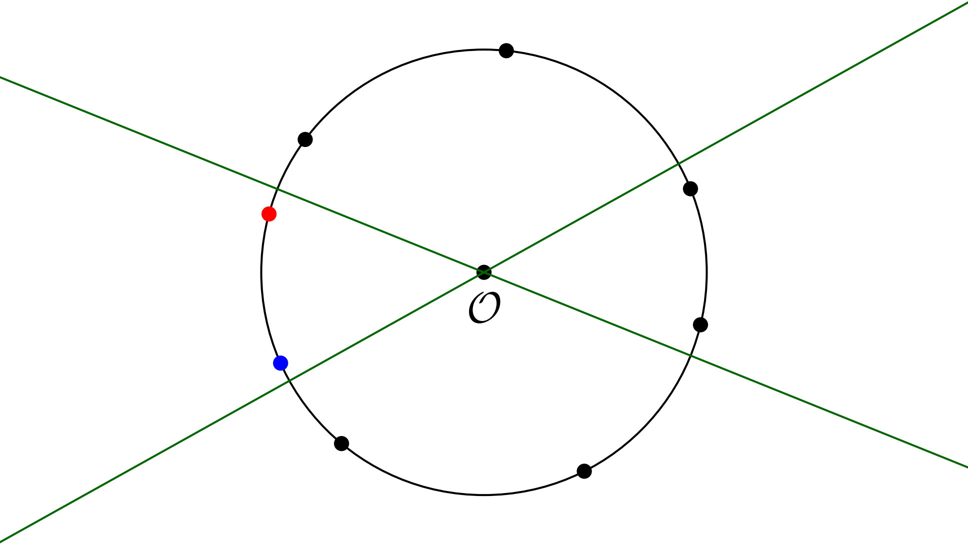
Hyperplane LSH

Query



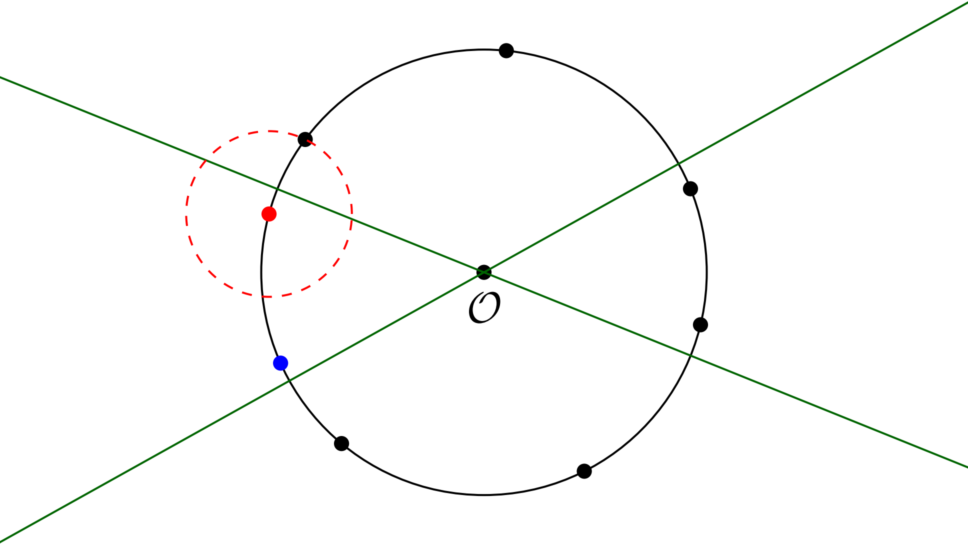
Hyperplane LSH

Collisions



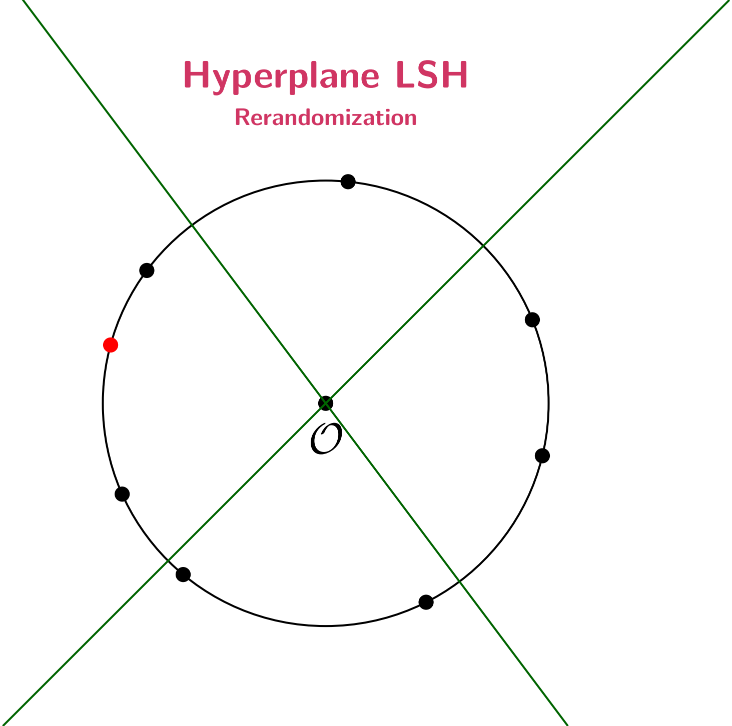
Hyperplane LSH

Failure



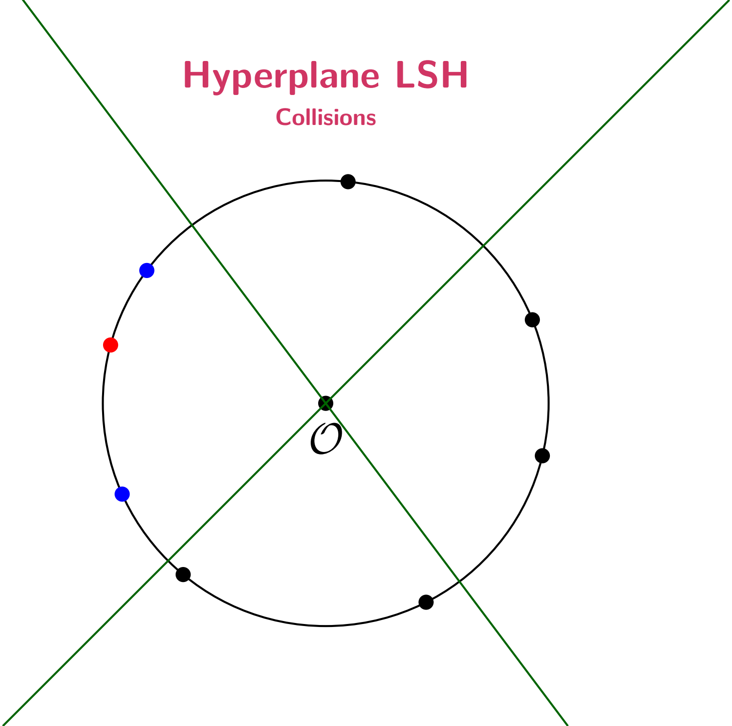
Hyperplane LSH

Rerandomization



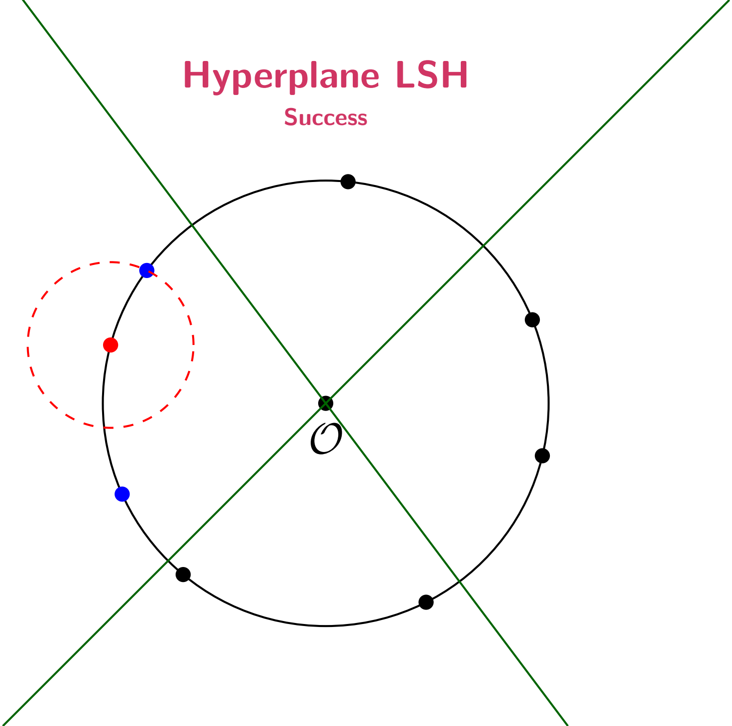
Hyperplane LSH

Collisions



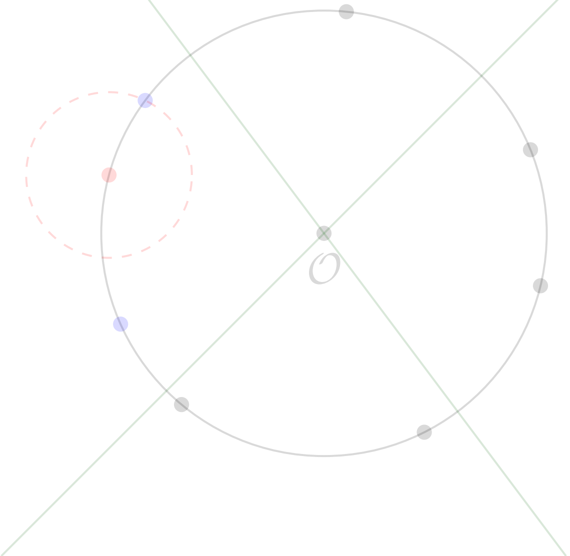
Hyperplane LSH

Success



Hyperplane LSH

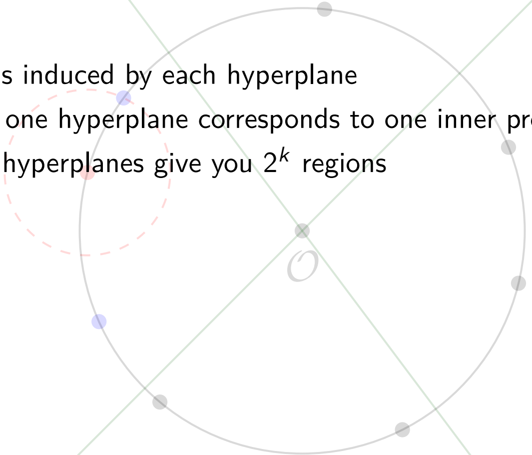
Overview



Hyperplane LSH

Overview

- 2 regions induced by each hyperplane
- Simple: one hyperplane corresponds to one inner product
- Fast: k hyperplanes give you 2^k regions



Hyperplane LSH

Overview

- 2 regions induced by each hyperplane
- Simple: one hyperplane corresponds to one inner product
- Fast: k hyperplanes give you 2^k regions

For "random" settings, query time $O(n^\rho)$ with

$$\rho = \frac{\sqrt{2}}{\pi \ln 2} \cdot \frac{1}{c} \left(1 + o_{d,c}(1)\right).$$

Hyperplane LSH

Overview

- 2 regions induced by each hyperplane
- Simple: one hyperplane corresponds to one inner product
- Fast: k hyperplanes give you 2^k regions

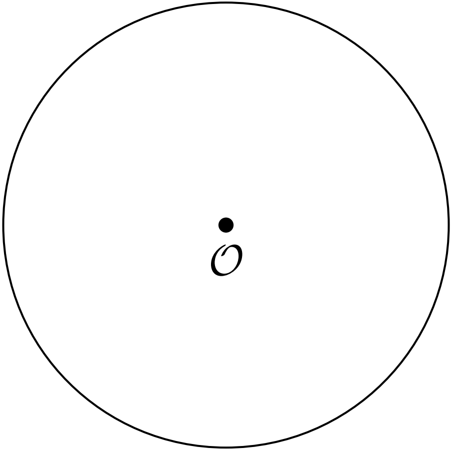
For "random" settings, query time $O(n^\rho)$ with

$$\rho = \frac{\sqrt{2}}{\pi \ln 2} \cdot \frac{1}{c} \left(1 + o_{d,c}(1)\right).$$

Efficient but suboptimal as $\rho \propto \frac{1}{c^2}$ is achievable

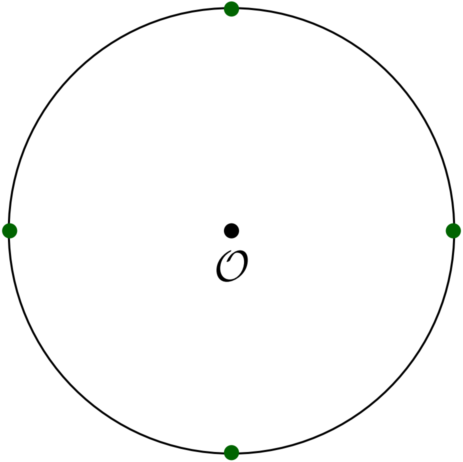
Cross-Polytope LSH

[Terasawa–Tanaka, WADS'07]
[Andoni et al., NIPS'15]



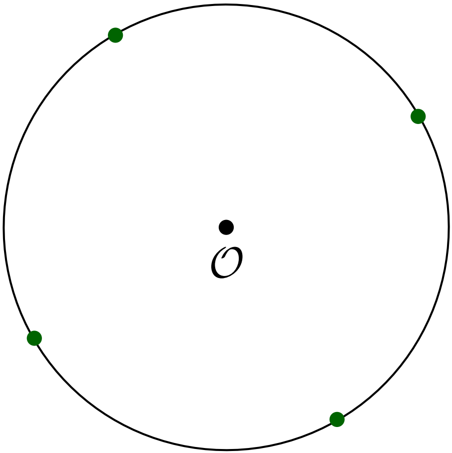
Cross-Polytope LSH

Vertices of cross-polytope (simplex)



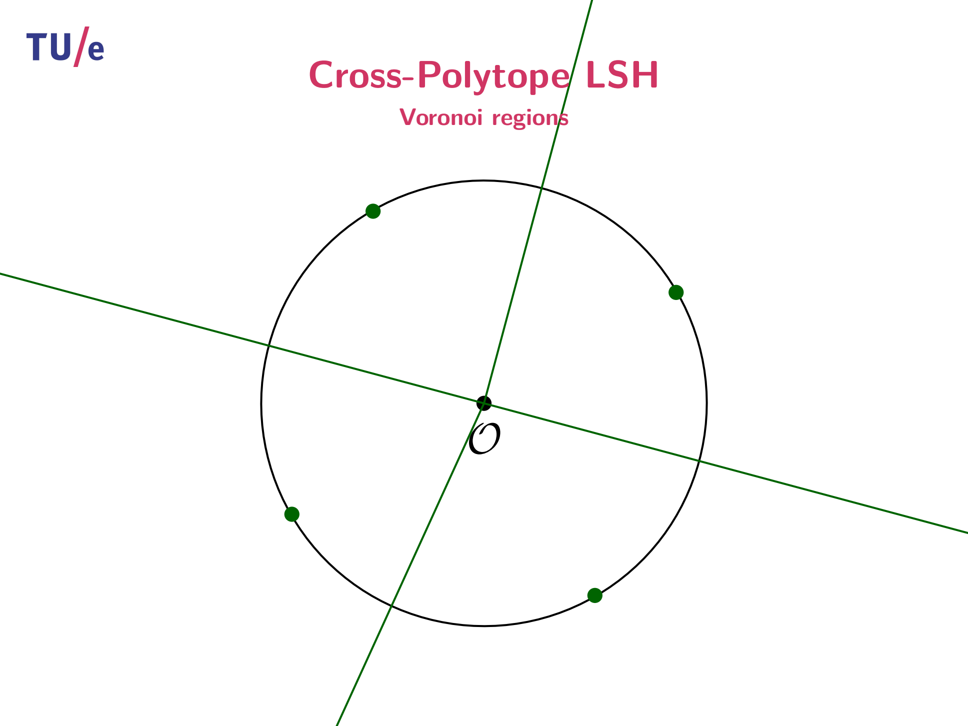
Cross-Polytope LSH

Random rotation



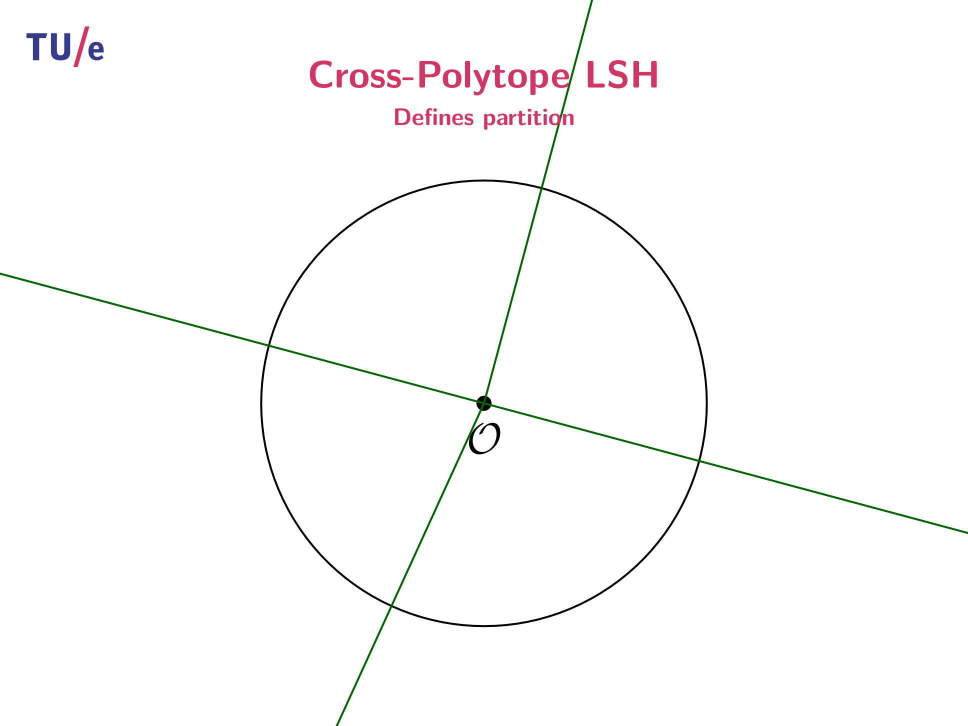
Cross-Polytope LSH

Voronoi regions



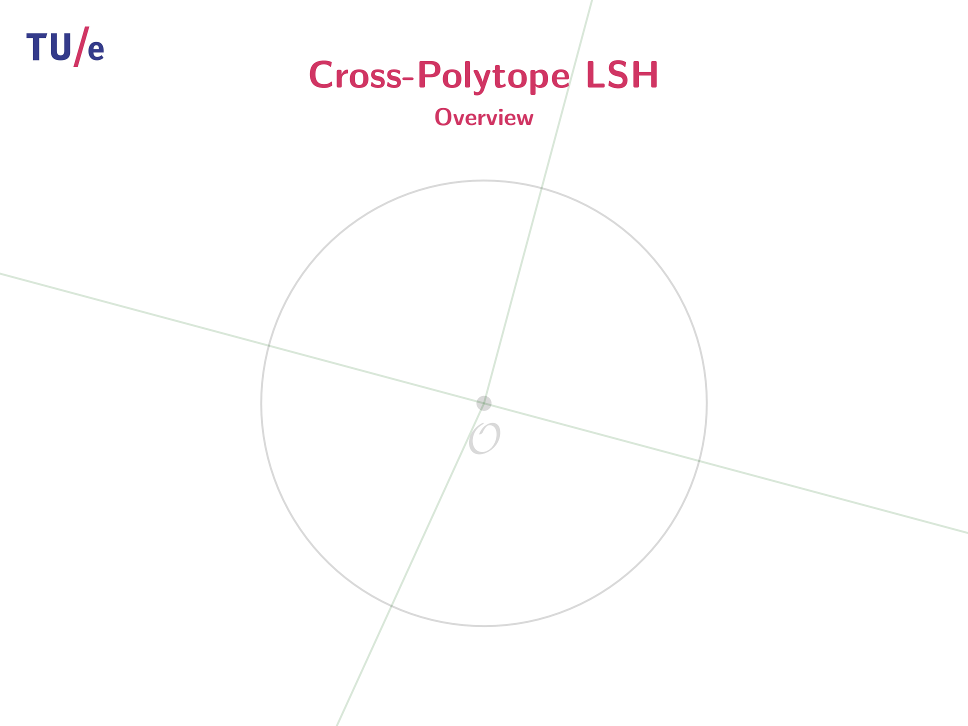
Cross-Polytope LSH

Defines partition



Cross-Polytope LSH

Overview



Cross-Polytope LSH

Overview

- $2d$ regions in d dimensions
- Advantage: regions same size and more symmetric

For "random" settings, query time $O(n^\rho)$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big)$$

Cross-Polytope LSH

Overview

- $2d$ regions in d dimensions
- Advantage: regions same size and more symmetric

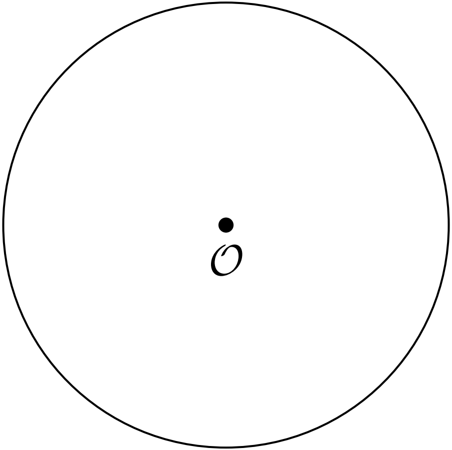
For "random" settings, query time $O(n^{\rho})$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big)$$

Essentially optimal for large c and $n = 2^{o(d)}$ [Dub'10, AR'15]

Spherical/Voronoi LSH

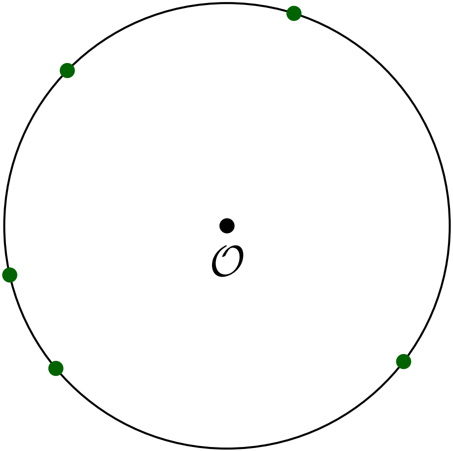
[Andoni et al., SODA'14]
[Andoni–Razenshteyn, STOC'15]



TU/e

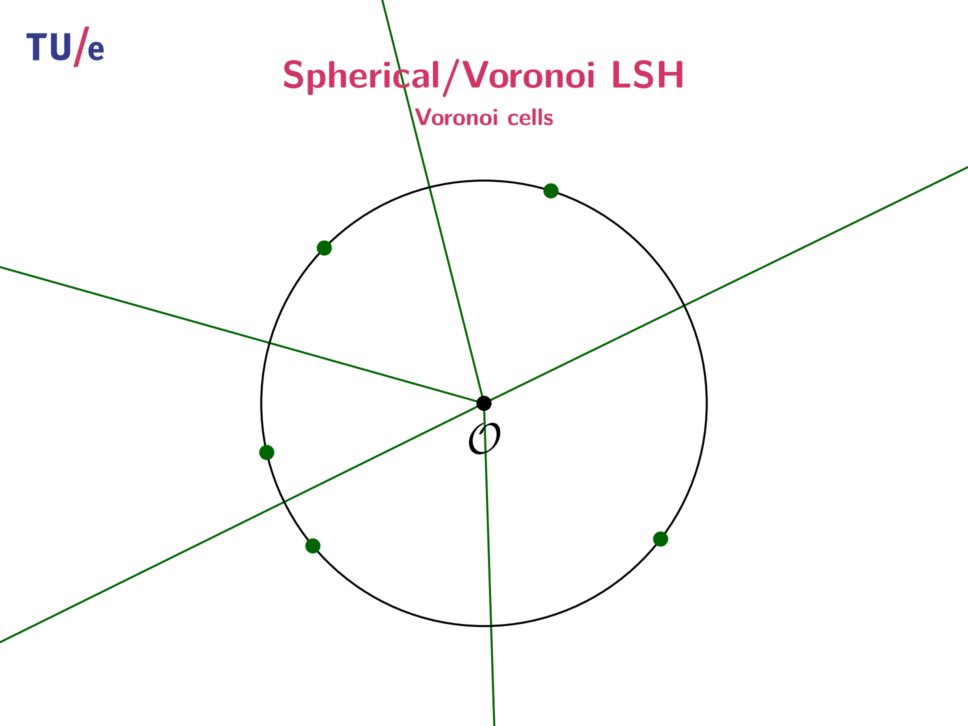
Spherical/Voronoi LSH

Random points



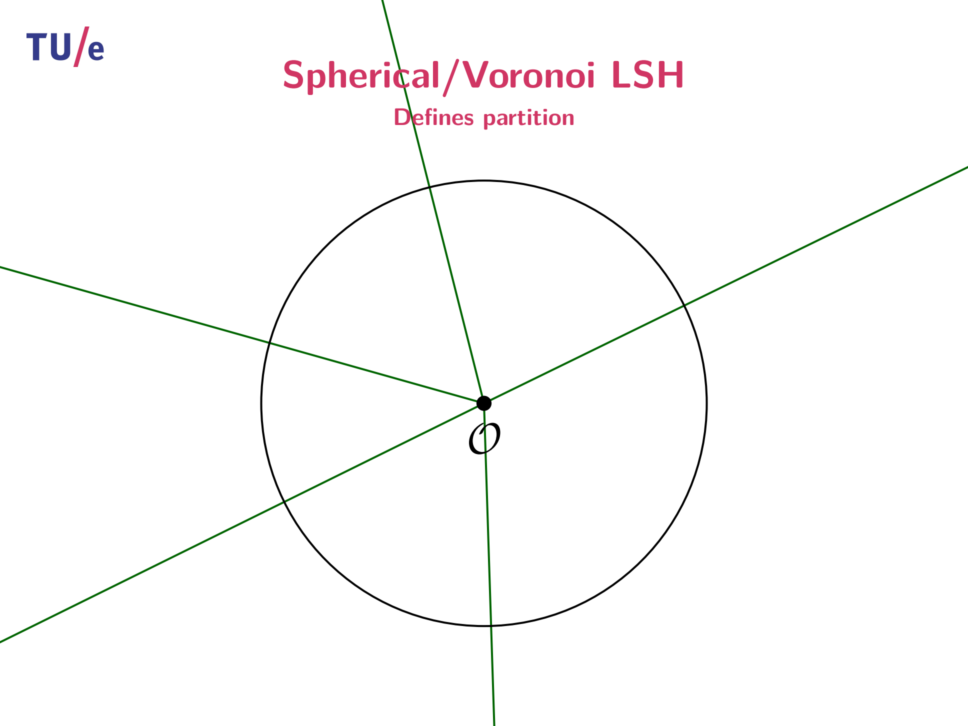
Spherical/Voronoi LSH

Voronoi cells



Spherical/Voronoi LSH

Defines partition



Spherical/Voronoi LSH

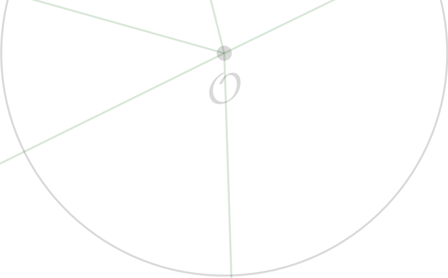
Overview

Spherical/Voronoi LSH

Overview

$2^{O(\sqrt{d})}$ points in d dimensions

- More points improves performance
- More points makes decoding slower



Spherical/Voronoi LSH

Overview

$2^{O(\sqrt{d})}$ points in d dimensions

- More points improves performance
- More points makes decoding slower

For "random" settings, query time $O(n^\rho)$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big).$$

TU/e

Spherical/Voronoi LSH

Overview

$2^{O(\sqrt{d})}$ points in d dimensions

- More points improves performance
- More points makes decoding slower

For "random" settings, query time $O(n^\rho)$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big).$$

Essentially optimal for large c and $n = 2^{o(d)}$

LSH overview

- Hyperplane LSH: 2 Voronoi cells
  - Efficient decoding
  - Suboptimal for large d, c
- Cross-Polytope LSH: $2d$ Voronoi cells
  - Reasonably efficient decoding
  - Optimal for large c and $n = 2^{o(d)}$
- Spherical/Voronoi LSH: $2^{O(\sqrt{d})}$ Voronoi cells
  - Slow decoding
  - Optimal for large c and $n = 2^{o(d)}$

LSH overview

- Hyperplane LSH: 2 Voronoi cells
  - Efficient decoding
  - Suboptimal for large d, c
- Cross-Polytope LSH: $2d$ Voronoi cells
  - Reasonably efficient decoding
  - Optimal for large c and $n = 2^{o(d)}$
- Spherical/Voronoi LSH: $2^{O(\sqrt{d})}$ Voronoi cells
  - Slow decoding
  - Optimal for large c and $n = 2^{o(d)}$

1. Can we use even more Voronoi cells?

LSH overview

- Hyperplane LSH: 2 Voronoi cells
  - Efficient decoding
  - Suboptimal for large d, c
- Cross-Polytope LSH: $2d$ Voronoi cells
  - Reasonably efficient decoding
  - Optimal for large c and $n = 2^{o(d)}$
- Spherical/Voronoi LSH: $2^{O(\sqrt{d})}$ Voronoi cells
  - Slow decoding
  - Optimal for large c and $n = 2^{o(d)}$

1. Can we use even more Voronoi cells?
2. Can decoding be made faster?

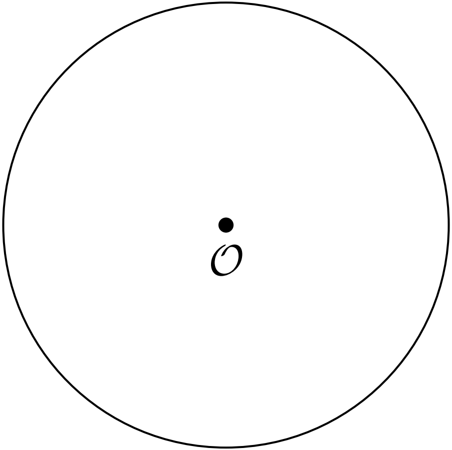
LSH overview

- Hyperplane LSH: 2 Voronoi cells
  - Efficient decoding
  - Suboptimal for large d, c
- Cross-Polytope LSH: $2d$ Voronoi cells
  - Reasonably efficient decoding
  - Optimal for large c and $n = 2^{o(d)}$
- Spherical/Voronoi LSH: $2^{O(\sqrt{d})}$ Voronoi cells
  - Slow decoding
  - Optimal for large c and $n = 2^{o(d)}$

1. Can we use even more Voronoi cells?
2. Can decoding be made faster?
3. What about $n = 2^{\Omega(d)}$?

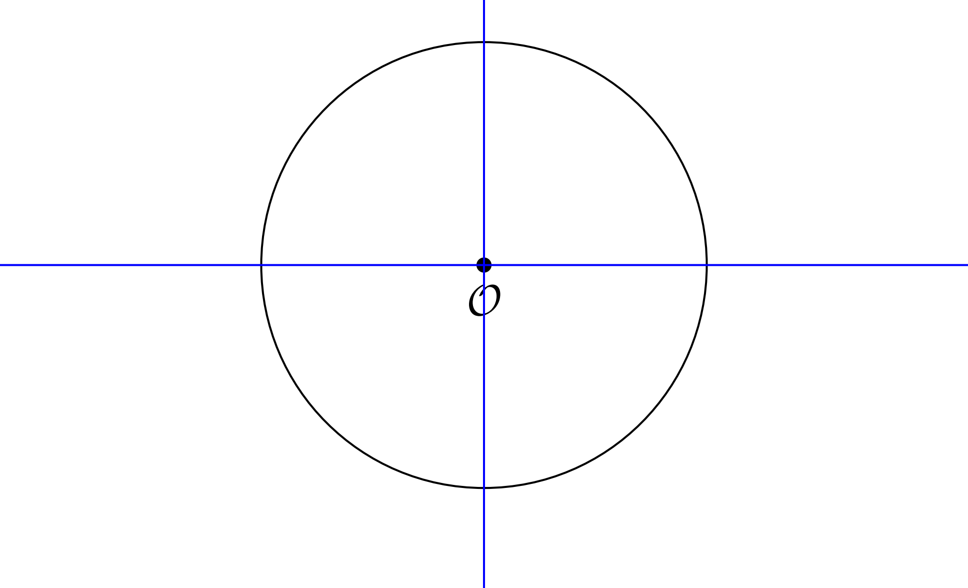
Structured filters

Overview



Structured filters

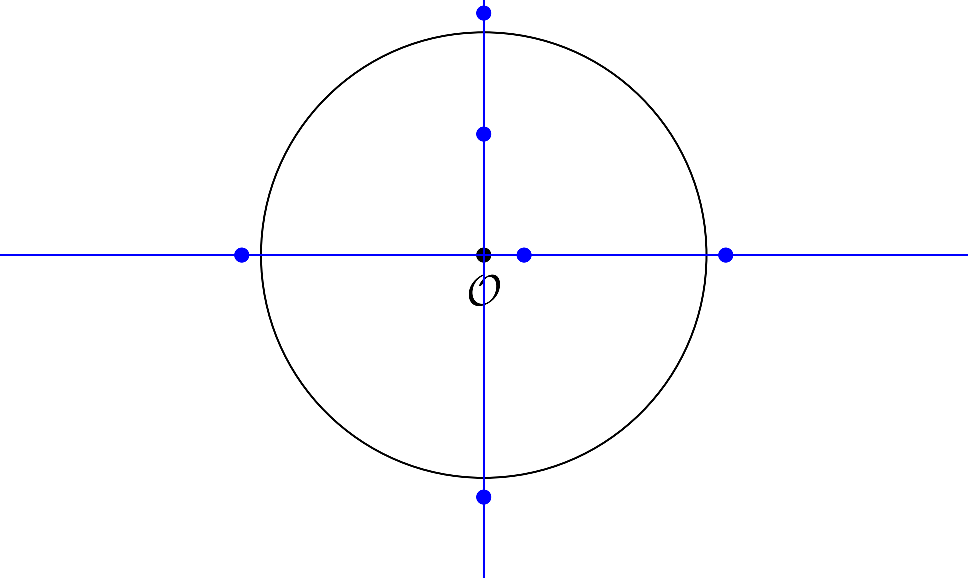
Partition dimensions into blocks



TU/e

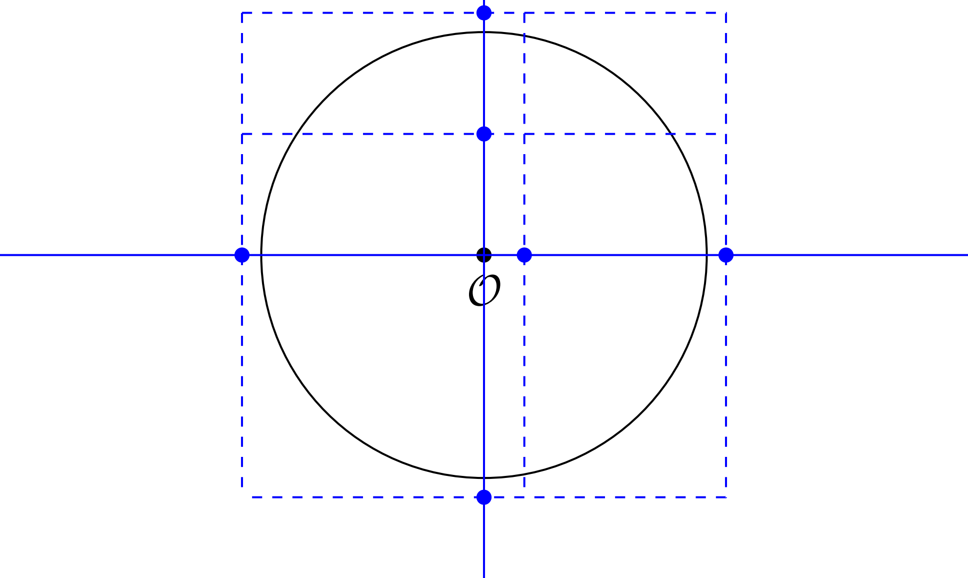
Structured filters

Random subcodes



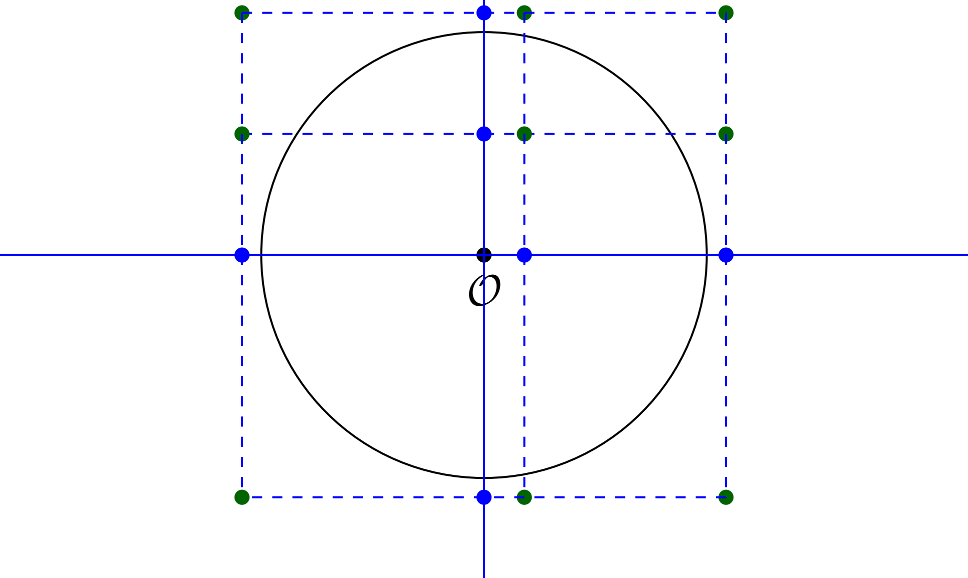
Structured filters

Construct concatenated code



Structured filters

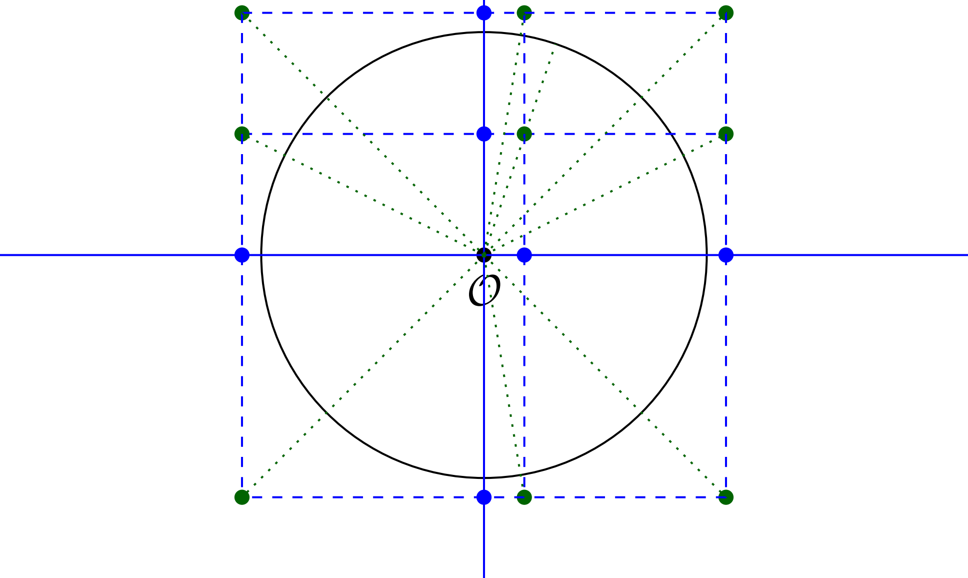
Construct concatenated code



TU/e

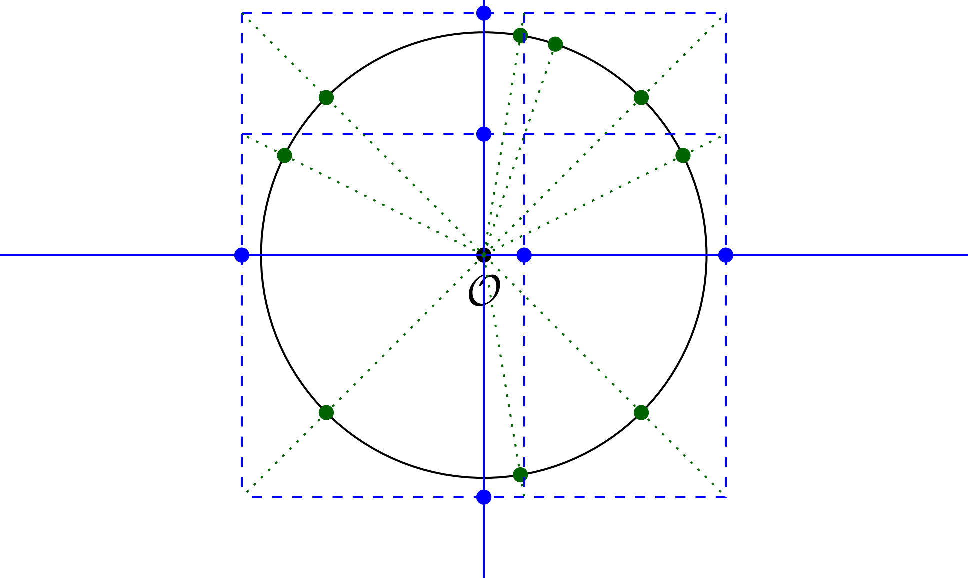
Structured filters

Normalize (only for example)



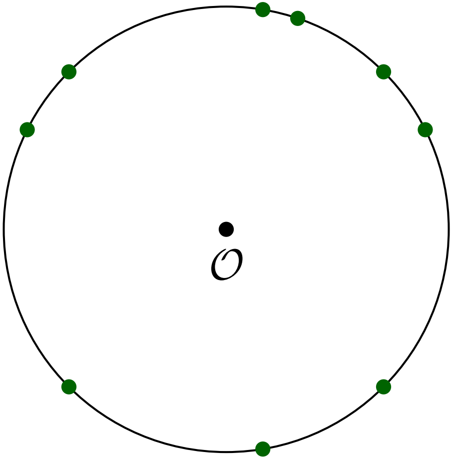
Structured filters

Normalize (only for example)



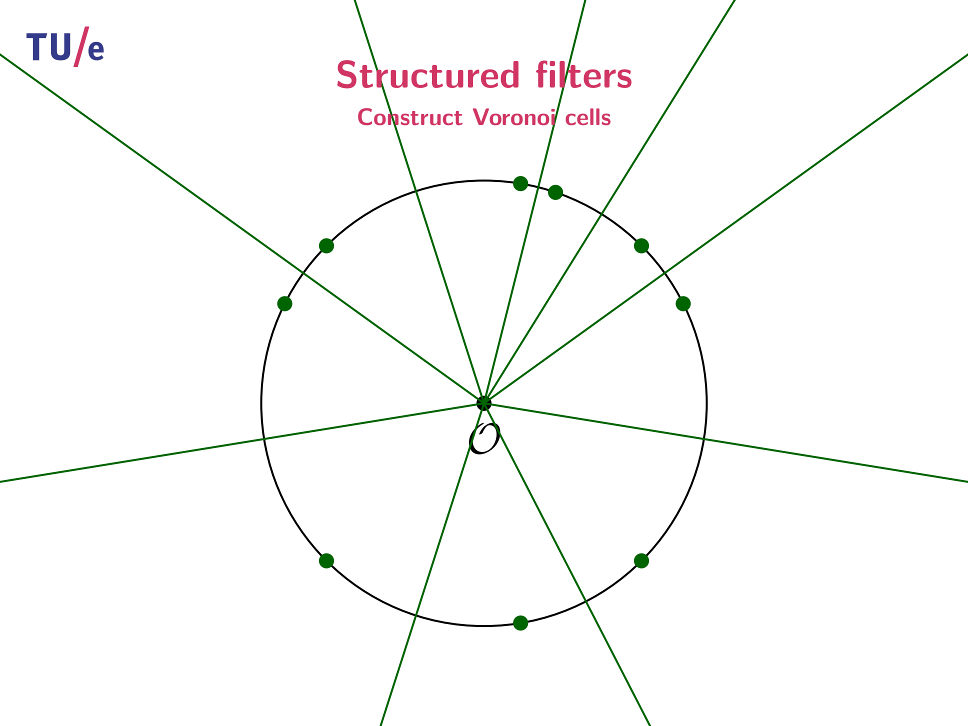
Structured filters

Normalize (only for example)



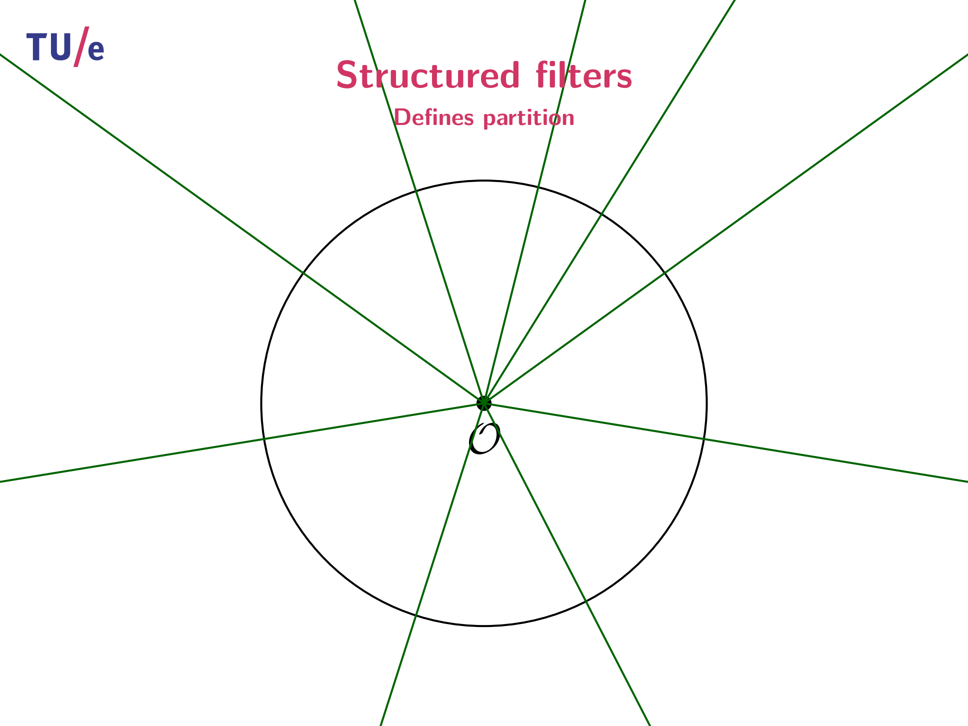
Structured filters

Construct Voronoi cells



Structured filters

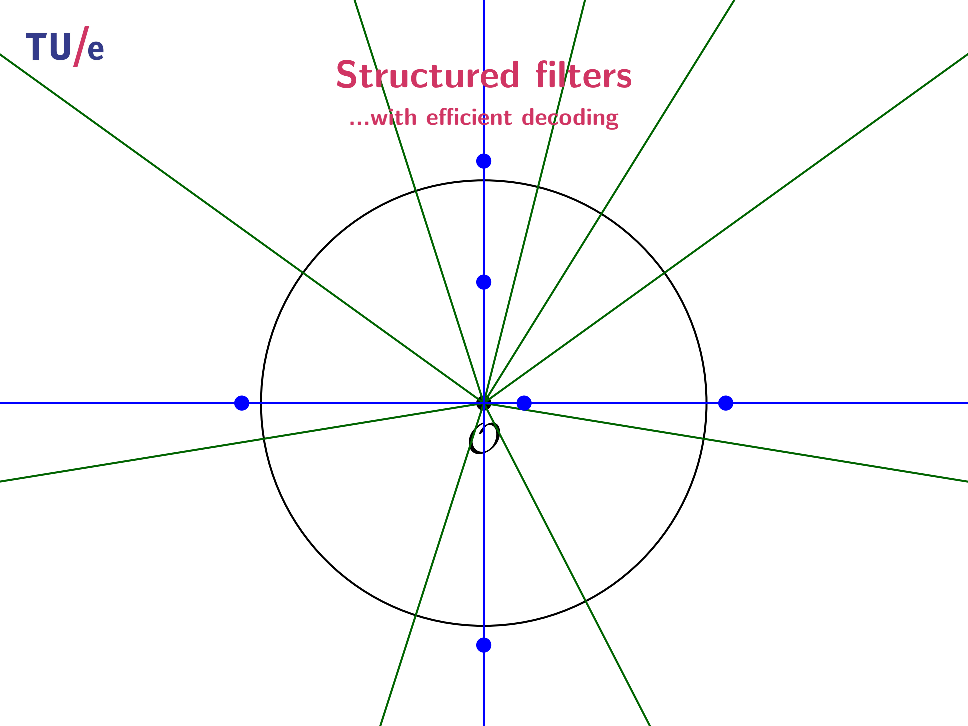
Defines partition



TU/e

Structured filters

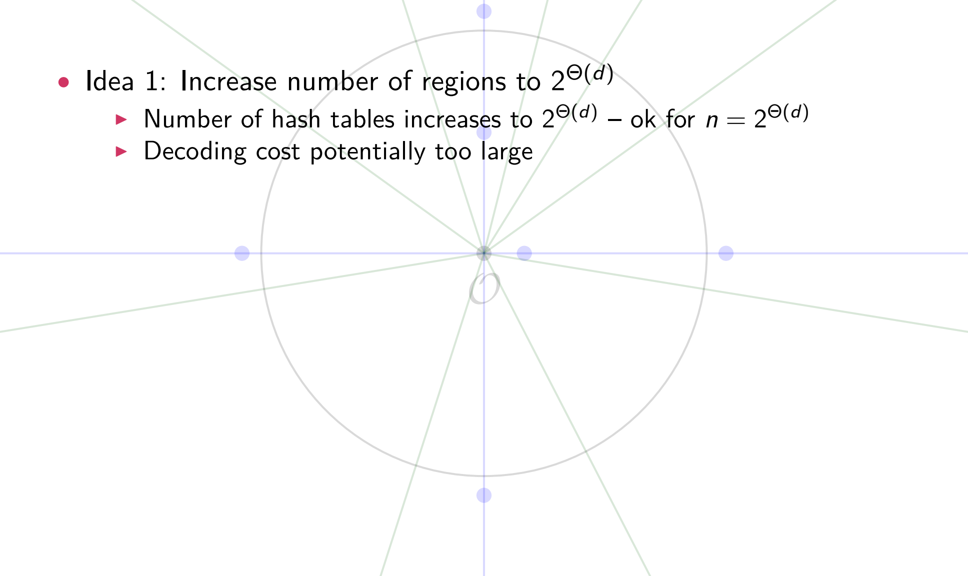
...with efficient decoding



Structured filters

Techniques

- Idea 1: Increase number of regions to $2^{\Theta(d)}$
  - Number of hash tables increases to $2^{\Theta(d)}$ – ok for $n = 2^{\Theta(d)}$
  - Decoding cost potentially too large



Structured filters

Techniques

- Idea 1: Increase number of regions to $2^{\Theta(d)}$
  - Number of hash tables increases to $2^{\Theta(d)}$ – ok for $n = 2^{\Theta(d)}$
  - Decoding cost potentially too large
- Idea 2: Use structured codes for random regions
  - Spherical/Voronoi LSH with dependent random points
  - Concatenated code of $\log d$ low-dim. spherical codes
  - Allows for efficient list-decoding

Structured filters

Techniques

- Idea 1: Increase number of regions to $2^{\Theta(d)}$
  - Number of hash tables increases to $2^{\Theta(d)}$ – ok for $n = 2^{\Theta(d)}$
  - Decoding cost potentially too large
- Idea 2: Use structured codes for random regions
  - Spherical/Voronoi LSH with dependent random points
  - Concatenated code of $\log d$ low-dim. spherical codes
  - Allows for efficient list-decoding
- Idea 3: Replace partitions with filters
  - Relaxation: filters need not partition the space
  - Simplified analysis
  - Might not be needed to achieve improvement

TU/e

Structured filters

Results

For random sparse settings ($n = 2^{o(d)}$), query time $O(n^\rho)$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big).$$

Structured filters

Results

For random sparse settings ($n = 2^{o(d)}$), query time $O(n^\rho)$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big).$$

For random dense settings ($n = 2^{\kappa d}$ with small κ), we obtain

$$\rho = \frac{1 - \kappa}{2c^2 - 1}\Big(1 + o_{d,\kappa}(1)\Big).$$

Structured filters

Results

For random sparse settings ($n = 2^{o(d)}$), query time $O(n^\rho)$ with

$$\rho = \frac{1}{2c^2 - 1}\Big(1 + o_d(1)\Big).$$

For random dense settings ($n = 2^{\kappa d}$ with small κ), we obtain

$$\rho = \frac{1 - \kappa}{2c^2 - 1}\Big(1 + o_{d,\kappa}(1)\Big).$$

For random dense settings ($n = 2^{\kappa d}$ with large κ), we obtain

$$\rho = \frac{-1}{2\kappa} \log\left(1 - \frac{1}{2c^2 - 1}\right) \Big(1 + o_d(1)\Big).$$

Asymmetric nearest neighbors

Previous results: symmetric NNS

- Query time: $O(n^\rho)$
- Update time: $O(n^\rho)$
- Preprocessing time: $O(n^{1+\rho})$
- Space complexity: $O(n^{1+\rho})$

Asymmetric nearest neighbors

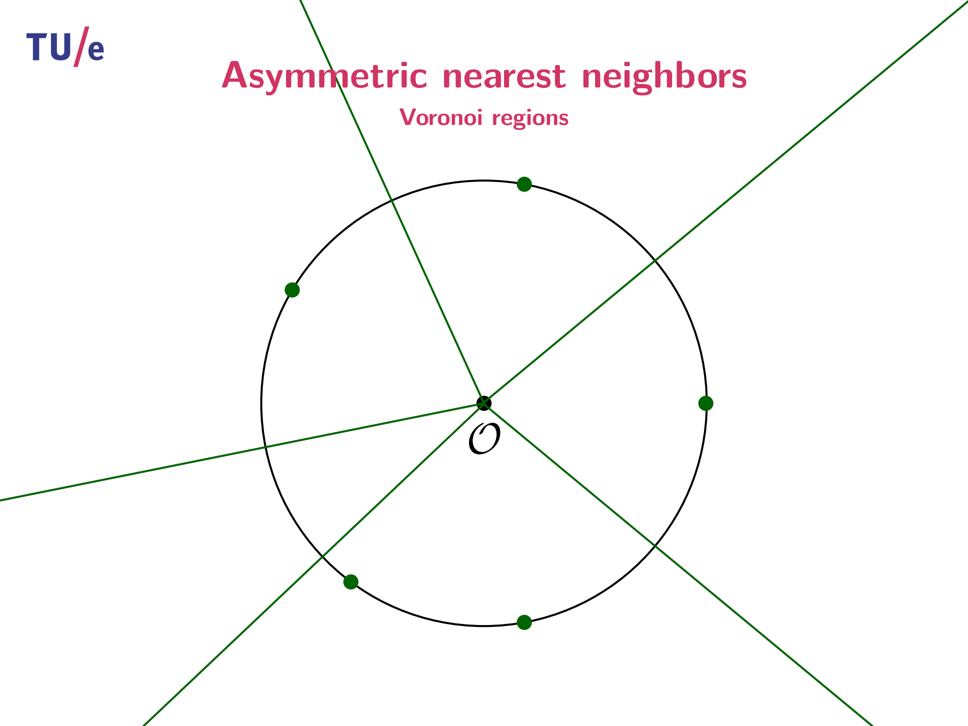
Previous results: symmetric NNS

- Query time: $O(n^{\rho})$
- Update time: $O(n^{\rho})$
- Preprocessing time: $O(n^{1+\rho})$
- Space complexity: $O(n^{1+\rho})$

Can we get a tradeoff between these costs?

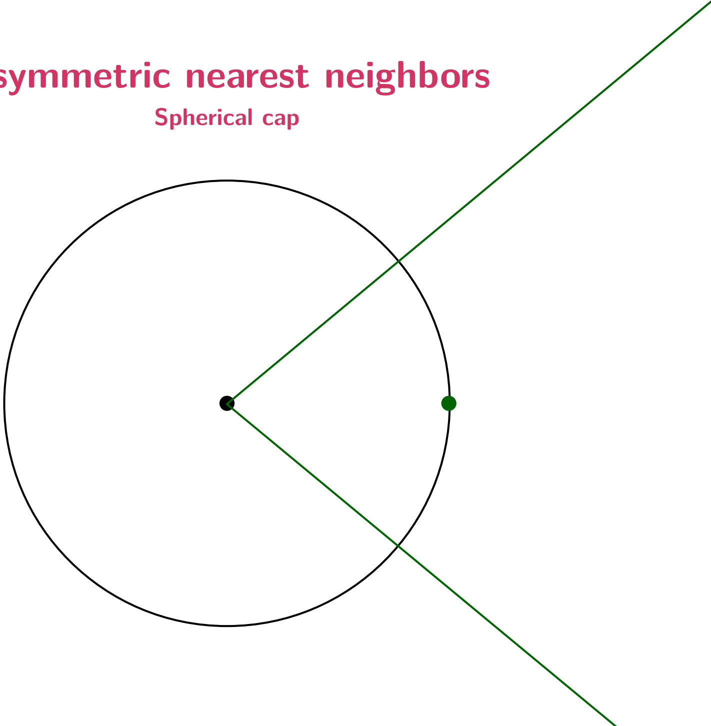
Asymmetric nearest neighbors

Voronoi regions



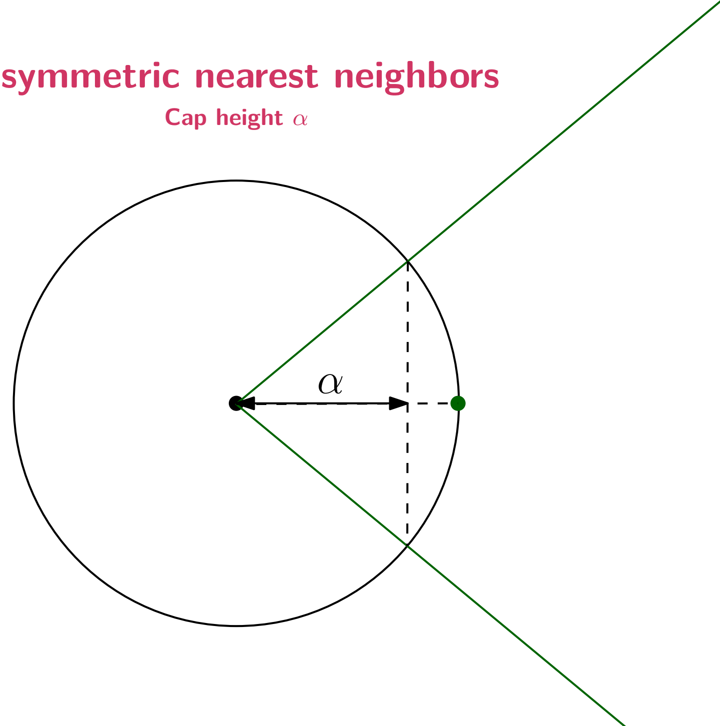
Asymmetric nearest neighbors

Spherical cap



Asymmetric nearest neighbors

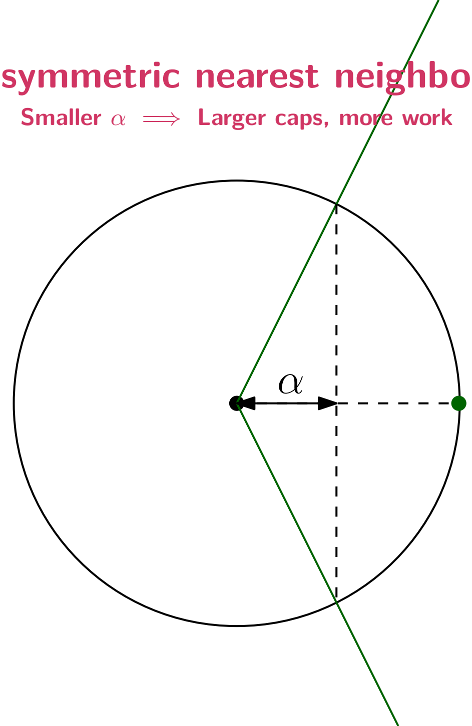
Cap height α



TU/e

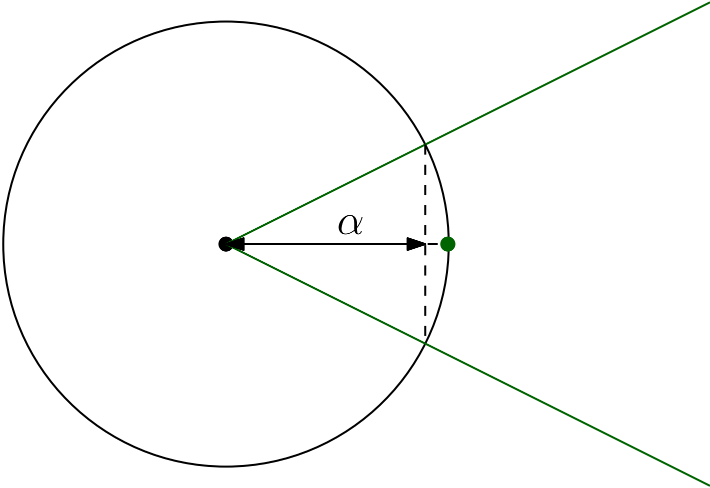
Asymmetric nearest neighbors

Smaller $\alpha \implies$ Larger caps, more work



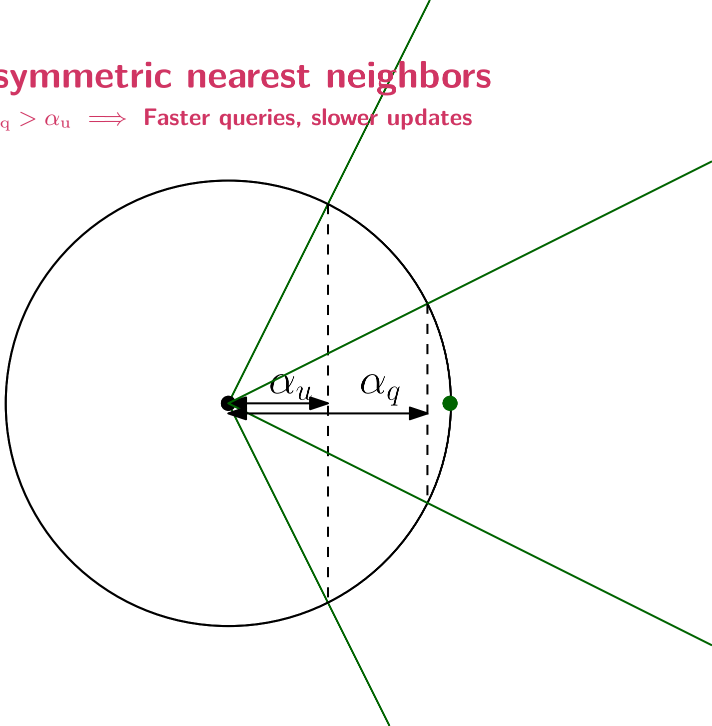
Asymmetric nearest neighbors

Larger $\alpha \implies$ Smaller caps, less work



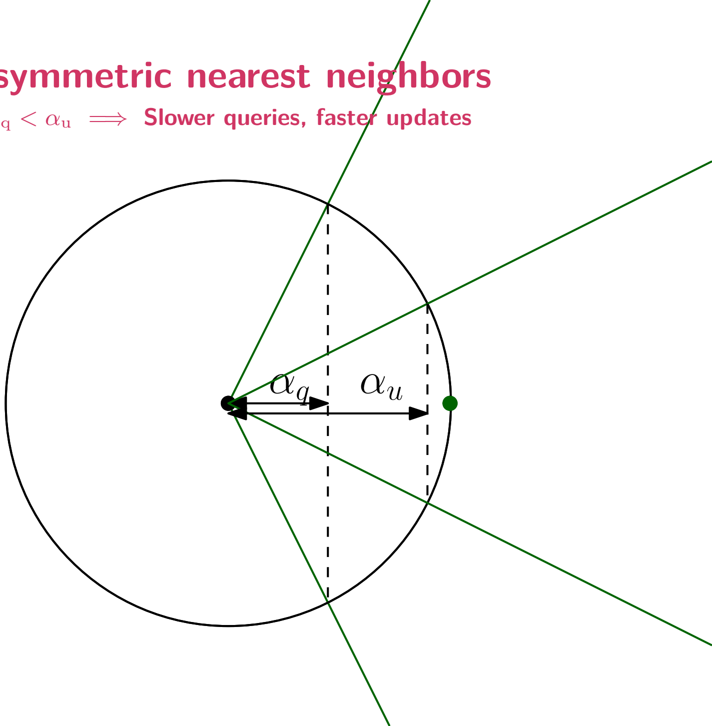
Asymmetric nearest neighbors

$\alpha_\mathrm{q} > \alpha_\mathrm{u} \implies$ **Faster queries, slower updates**



Asymmetric nearest neighbors

$\alpha_{\mathrm{q}} < \alpha_{\mathrm{u}} \implies$ **Slower queries, faster updates**



Asymmetric nearest neighbors

Results

	General expressions
Minimize space	$\rho_{\mathrm{q}} = (2c^2 - 1)/c^4$
$(\alpha_{\mathrm{q}}/\alpha_{\mathrm{u}} = \cos\theta)$	$\rho_{\mathrm{u}} = 0$
Balance costs	$\rho_{\mathrm{q}} = 1/(2c^2 - 1)$
$(\alpha_{\mathrm{q}}/\alpha_{\mathrm{u}} = 1)$	$\rho_{\mathrm{u}} = 1/(2c^2 - 1)$
Minimize time	$\rho_{\mathrm{q}} = 0$
$(\alpha_{\mathrm{q}}/\alpha_{\mathrm{u}} = 1/\cos\theta)$	$\rho_{\mathrm{u}} = (2c^2 - 1)/(c^2 - 1)^2$

Query time $O(n^{\rho_{\mathrm{q}}})$, update time $O(n^{\rho_{\mathrm{u}}})$, preprocessing time $O(n^{1+\rho_{\mathrm{u}}})$, space complexity $O(n^{1+\rho_{\mathrm{u}}})$

Asymmetric nearest neighbors

Results

	General expressions	Small $c = 1 + \varepsilon$
Minimize space	$\rho_q = (2c^2 - 1)/c^4$	$\rho_q = 1 - 4\varepsilon^2 + O(\varepsilon^3)$
$(\alpha_q/\alpha_u = \cos\theta)$	$\rho_u = 0$	$\rho_u = 0$
Balance costs	$\rho_q = 1/(2c^2 - 1)$	$\rho_q = 1 - 4\varepsilon + O(\varepsilon^2)$
$(\alpha_q/\alpha_u = 1)$	$\rho_u = 1/(2c^2 - 1)$	$\rho_u = 1 - 4\varepsilon + O(\varepsilon^2)$
Minimize time	$\rho_q = 0$	$\rho_q = 0$
$(\alpha_q/\alpha_u = 1/\cos\theta)$	$\rho_u = (2c^2 - 1)/(c^2 - 1)^2$	$\rho_u = 1/(4\varepsilon^2) + O(1/\varepsilon)$

Query time $O(n^{\rho_q})$, update time $O(n^{\rho_u})$, preprocessing time $O(n^{1+\rho_u})$, space complexity $O(n^{1+\rho_u})$

Asymmetric nearest neighbors

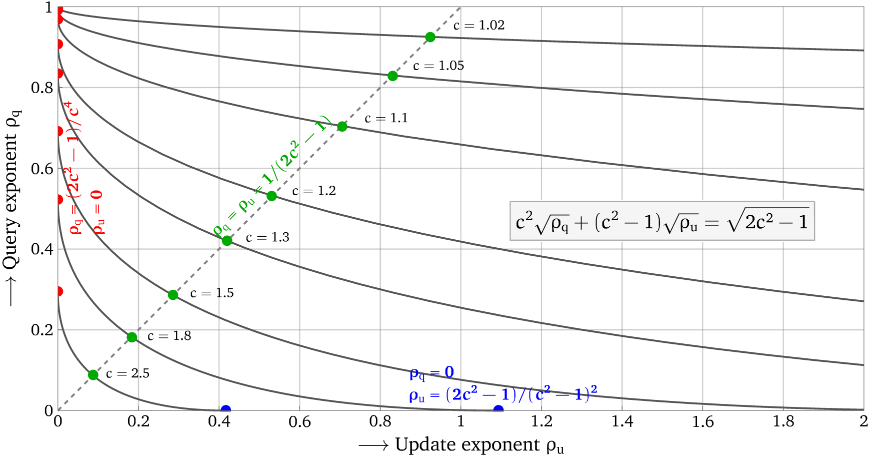
Results

	General expressions	Large $c \to \infty$
Minimize space $(\alpha_q/\alpha_u = \cos\theta)$	$\rho_q = (2c^2 - 1)/c^4$ $\rho_u = 0$	$\rho_q = 2/c^2 + O(1/c^4)$ $\rho_u = 0$
Balance costs $(\alpha_q/\alpha_u = 1)$	$\rho_q = 1/(2c^2 - 1)$ $\rho_u = 1/(2c^2 - 1)$	$\rho_q = 1/(2c^2) + O(1/c^4)$ $\rho_u = 1/(2c^2) + O(1/c^4)$
Minimize time $(\alpha_q/\alpha_u = 1/\cos\theta)$	$\rho_q = 0$ $\rho_u = (2c^2 - 1)/(c^2 - 1)^2$	$\rho_q = 0$ $\rho_u = 2/c^2 + O(1/c^4)$

Query time $O(n^{\rho_q})$, update time $O(n^{\rho_u})$, preprocessing time $O(n^{1+\rho_u})$, space complexity $O(n^{1+\rho_u})$

Asymmetric nearest neighbors

Tradeoffs



Conclusions

Main result: Allow using more regions with list-decodable codes

- For $n = 2^{o(d)}$, non-asymptotic improvement
- For $n = 2^{\Theta(d)}$, asymptotic improvement
- Corollary: Lower bounds for $n = 2^{o(d)}$ do not hold for $n = 2^{\Theta(d)}$
- Improved tradeoffs between query and update complexities

Conclusions

Main result: Allow using more regions with list-decodable codes

- For $n = 2^{o(d)}$, non-asymptotic improvement
- For $n = 2^{\Theta(d)}$, asymptotic improvement
- Corollary: Lower bounds for $n = 2^{o(d)}$ do not hold for $n = 2^{\Theta(d)}$
- Improved tradeoffs between query and update complexities

Open problems

- Tradeoff for $n = 2^{o(d)}$ optimal?
- Lower bounds for $n = 2^{\Theta(d)}$?
- Apply similar ideas to other norms?
- Practicality?

Conclusions

Main result: Allow using more regions with list-decodable codes

- For $n = 2^{o(d)}$, non-asymptotic improvement
- For $n = 2^{\Theta(d)}$, asymptotic improvement
- Corollary: Lower bounds for $n = 2^{o(d)}$ do not hold for $n = 2^{\Theta(d)}$
- Improved tradeoffs between query and update complexities

Open problems

- Tradeoff for $n = 2^{o(d)}$ optimal?
- Lower bounds for $n = 2^{\Theta(d)}$?
- Apply similar ideas to other norms?
- Practicality?

Questions?