

# **An Introduction to Geometric Data Structures and Dual-Tree Algorithms**

Ryan R. Curtin

September 25, 2018

# Introduction and Overview

For the next (approximate) hour, you will be subjected to:

- how distance-based problems appear in all kinds of machine learning problems,
- a detailed introduction to data structures like the *kd*-tree and others,
- a unifying generalization for these types of tree structures called ‘space trees’,
- a dive into how single-tree and dual-tree algorithms can be used to answer distance-based problems,
- an incomplete list of the problems we can solve this way,
- some theoretical concerns,
- some empirical results that show it was all worthwhile,
- and some incomplete thoughts about the future.

# Distance-based Problems

It turns out that lots of machine learning problems depend on distances, either obviously or non-obviously.

- **$k$ -nearest-neighbor search**: find the  $k$  points with minimum  $d(\cdot, \cdot)$  to the query point(s)
- **range search**: find the points with  $d(\cdot, \cdot)$  in a given range from the query
- **kernel density estimation**: compute a density estimate that depends on  $d(\cdot, \cdot)$  between the query location and all points
- **max-kernel search**: find the point with maximum similarity (implicitly depends on a distance  $d(\cdot, \cdot)$ )
- **$k$ -means clustering**: iterative algorithm depending on distances
- **DBSCAN clustering**: includes a range search query
- **collaborative filtering**: often includes a  $k$ -NN step
- **single-linkage clustering**: equivalent to computing an MST on distances
- **manifold learning algorithms**: usually an optimization constrained on distance computations between points, often related to  $k$ -NN
- **Gaussian mixture models**: fits depend on distance between points
- **Hidden Markov model training**: computation of probabilities depends on distances  $d(\cdot, \cdot)$ .
- and so on...

# Only two problems

Let's ensmallen the universe.

In our temporary universe  $\mathcal{U}_{\text{temp}}$ , there exist only two problems that are worth solving:

- $k$ -nearest-neighbor search
- range search

If we can solve these problems efficiently, we have determined the meaning of life in  $\mathcal{U}_{\text{temp}}$ !

# $k$ -nearest neighbor search

Given some *query point*  $p_q$  and some *reference set*  $S_r$ , find

$$k\text{-argmin}_{p_r \in S_r} d(p_q, p_r).$$

# Range search

Given some query point  $p_q$ , some range  $[l, u]$ , and a *reference set*  $S_r$ , find

$$\{p_r : p_r \in S_r, l \leq d(p_q, p_r) \leq u\}$$

(variant: *range count*)

# How do we solve these problems fast?

- Exact computation.

# How do we solve these problems fast?

- Exact computation. *not efficient...*



# How do we solve these problems fast?

- Exact computation. *not efficient...*
- Hashing or low-dimensional projections: make the problem 'simpler' and introduce some approximation.

# How do we solve these problems fast?

- Exact computation. *not efficient...*
- Hashing or low-dimensional projections: make the problem ‘simpler’ and introduce some approximation.
- Sampling or coresets: select some points from  $S_r$  that are likely to be good (approximate) solutions.

# How do we solve these problems fast?

- Exact computation. *not efficient...*
- Hashing or low-dimensional projections: make the problem ‘simpler’ and introduce some approximation.
- Sampling or coresets: select some points from  $S_r$  that are likely to be good (approximate) solutions.
- **Trees:** build an indexing structure on  $S_r$  for (hopefully) logarithmic-time search.

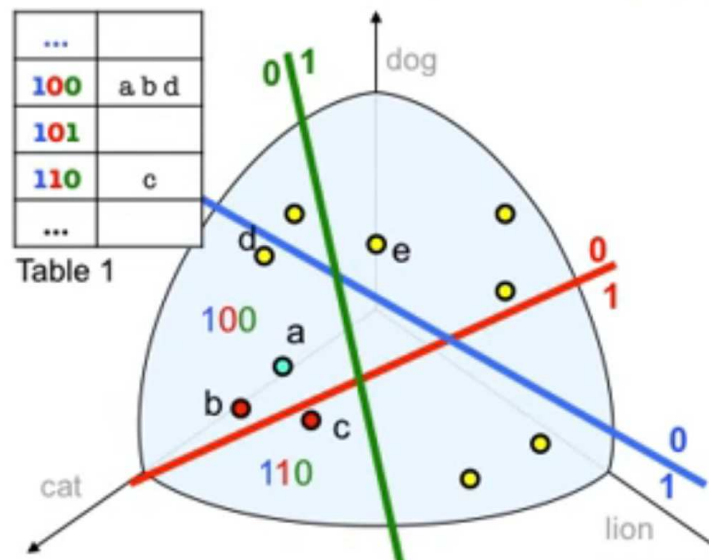
# How do we solve these problems fast?

- Exact computation. *not efficient...*
- Hashing or low-dimensional projections: make the problem ‘simpler’ and introduce some approximation.
- Sampling or coresets: select some points from  $S_r$  that are likely to be good (approximate) solutions.
- **Trees:** build an indexing structure on  $S_r$  for (hopefully) logarithmic-time search.
- Other approaches I won’t consider...

# Diversion: locality-sensitive hashing

Let's (very) briefly consider **locality-sensitive hashing for nearest neighbor search** because it's important and relevant.

- Pick some random hyperplane  $P_i$ .
- If  $p_r \in S_r$  is to the left of  $P_i$  then the hash code is 0; otherwise it is 1.
- Do this a lot of times to assemble a long hash code.
- When searching for a nearest neighbor of  $p_q$ , find the hash code of  $p_q$  and look at all points of  $S_r$  that have the same hash code.



Let's ignore all the theory about why this works, how well it works, etc., and just say it reasonably works and leave it at that.

# Diversion: locality-sensitive hashing

Let's (very) briefly consider **locality-sensitive hashing for nearest neighbor search** because it's important and relevant.

- Pick some random hyperplane  $P_i$ .
- If  $p_r \in S_r$  is to the left of  $P_i$  then the hash code is 0; otherwise it is 1.
- Do this a lot of times to assemble a long hash code.
- When searching for a nearest neighbor of  $p_q$ , find the hash code of  $p_q$  and look at all points of  $S_r$  that have the same hash code.



*other considerations: how many  $P_i$ ?; multiple bins for each  $P_i$ ; probe multiple nearby hash codes for similar  $p_q$ ; how to select  $P_i$ ?; connections to trees; hybrid approaches; multi-level hash tables; etc...*

Let's ignore all the theory about why this works, how well it works, etc., and just say it reasonably works and leave it at that.

# Un-diversion: trees

For the rest of the talk let's focus on trees. In our little  $\mathcal{U}_{\text{temp}}$  we care about

- Empirical speed
- Applicability to real-world situations

# Un-diversion: trees

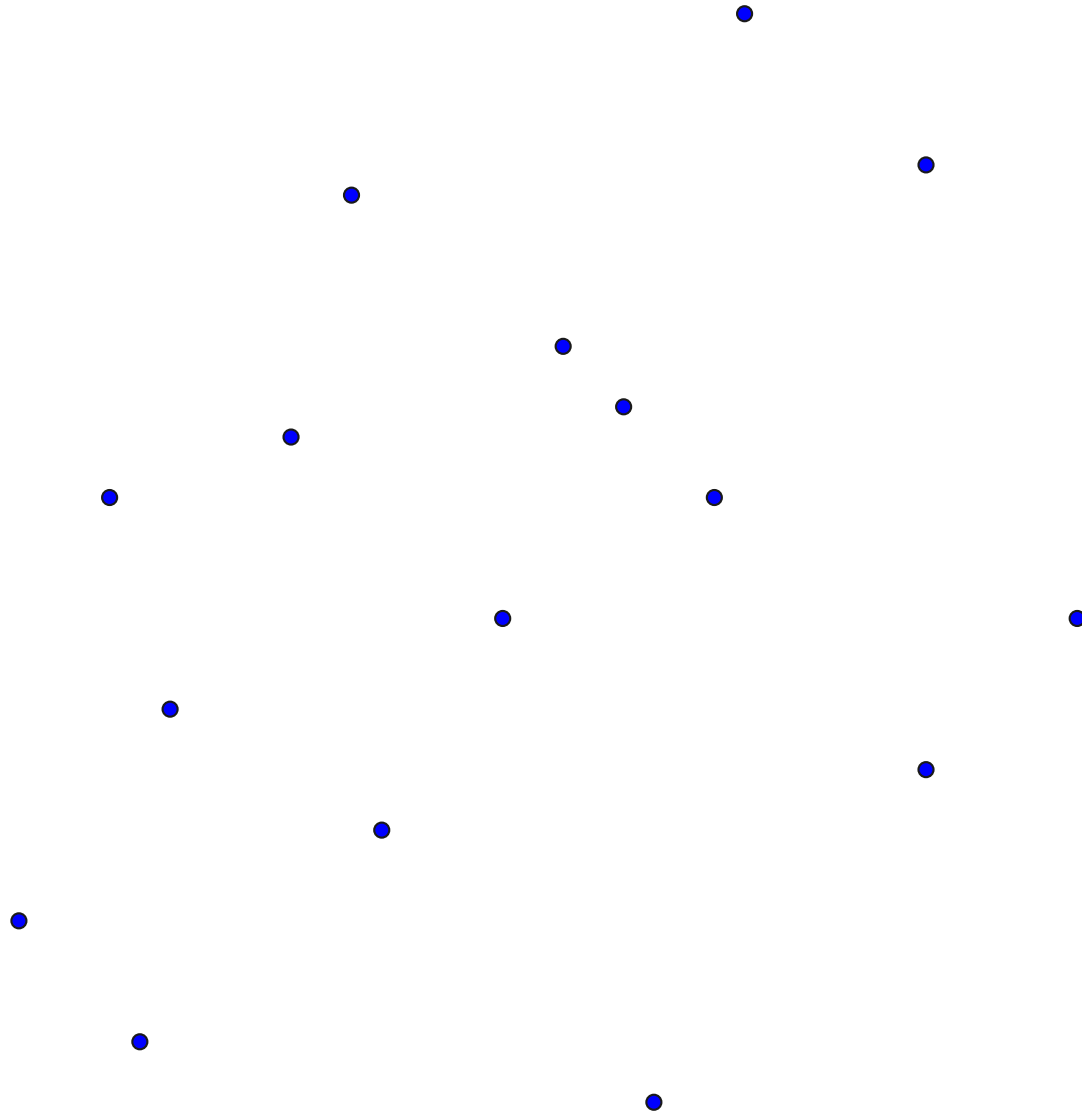
For the rest of the talk let's focus on trees. In our little  $\mathcal{U}_{\text{temp}}$  we care about

- Empirical speed
- Applicability to real-world situations

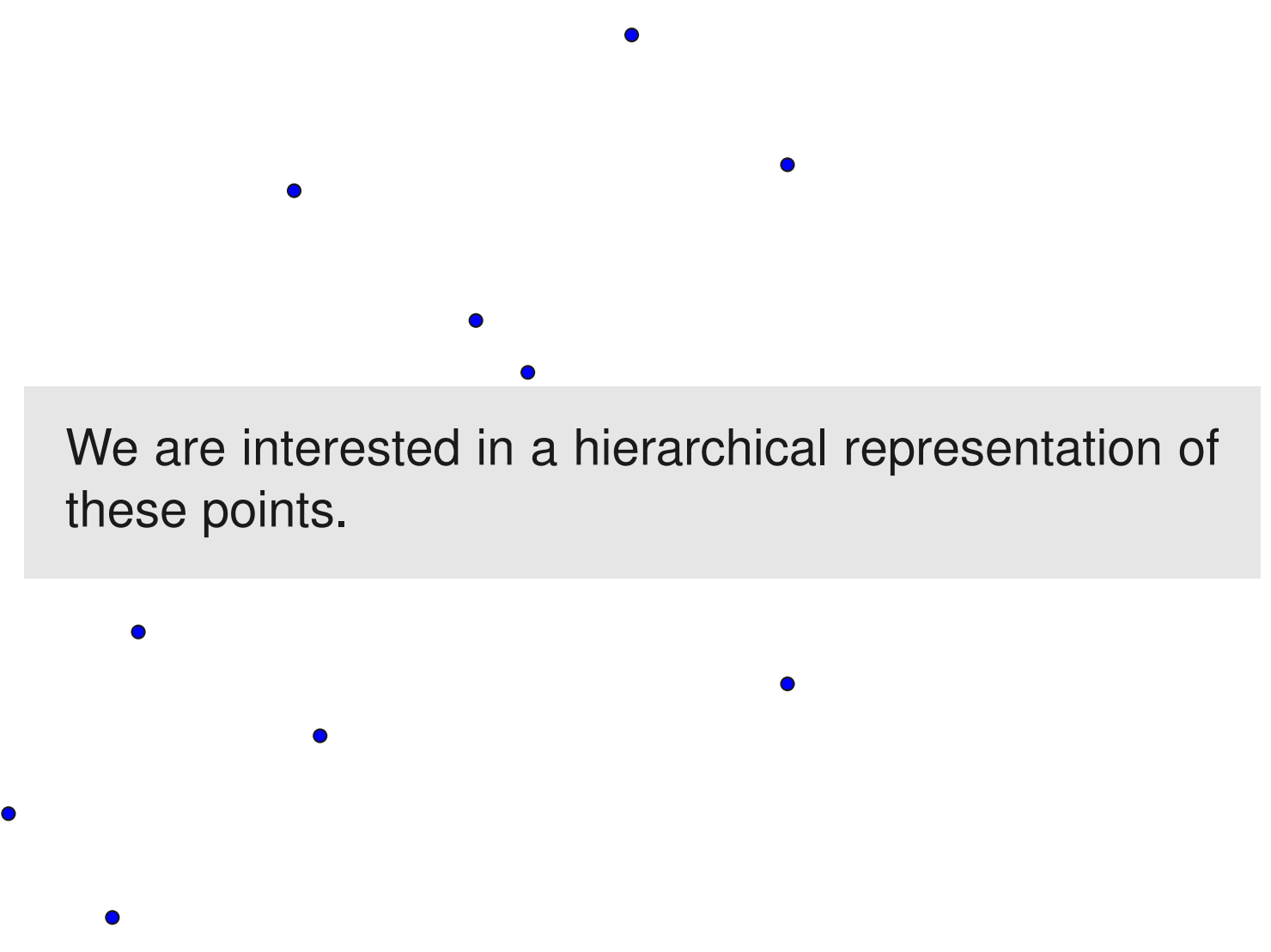
(Okay, theory is still good too, but it's secondary.)



# The $kd$ -tree

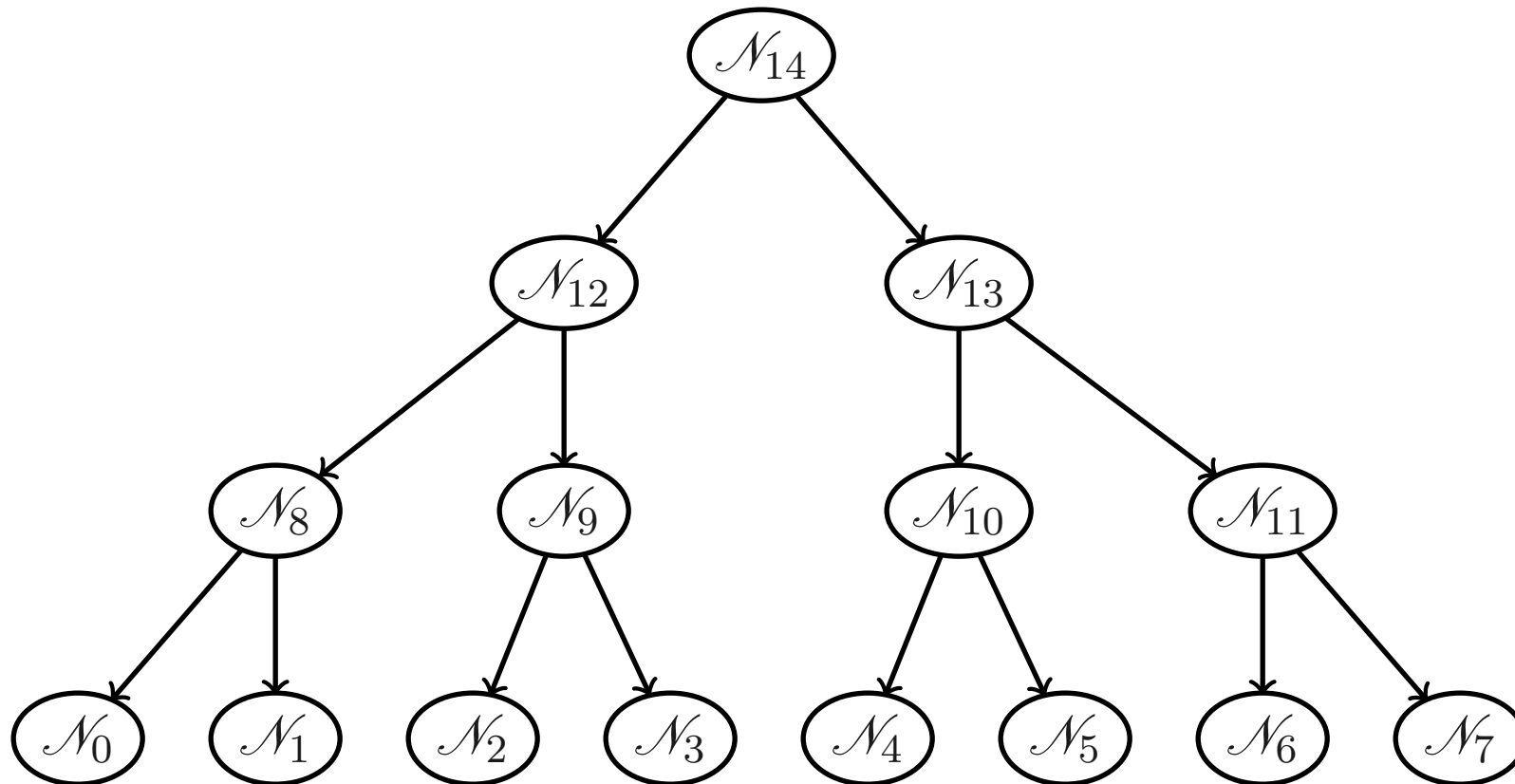


# The $kd$ -tree

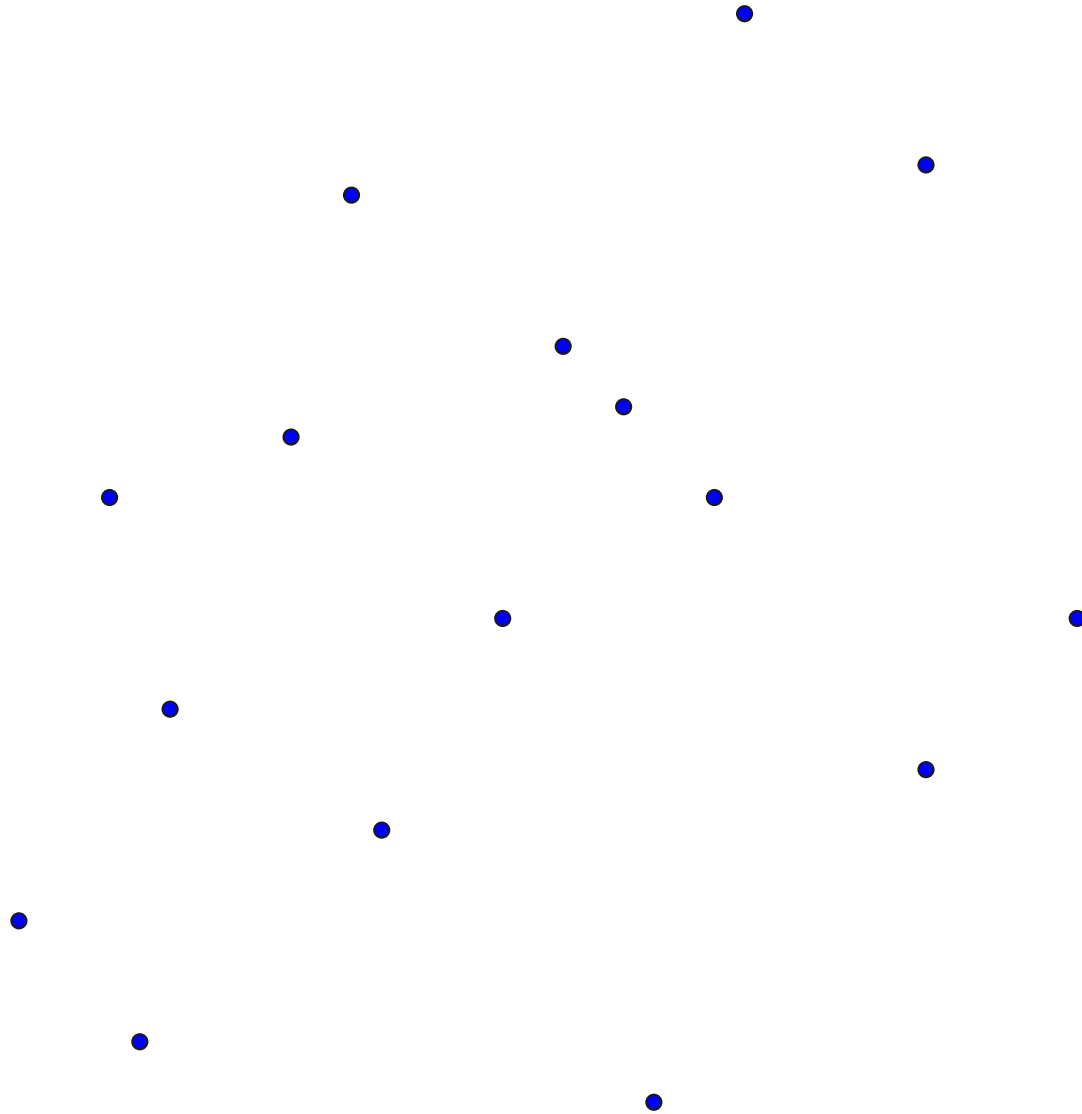


We are interested in a hierarchical representation of these points.

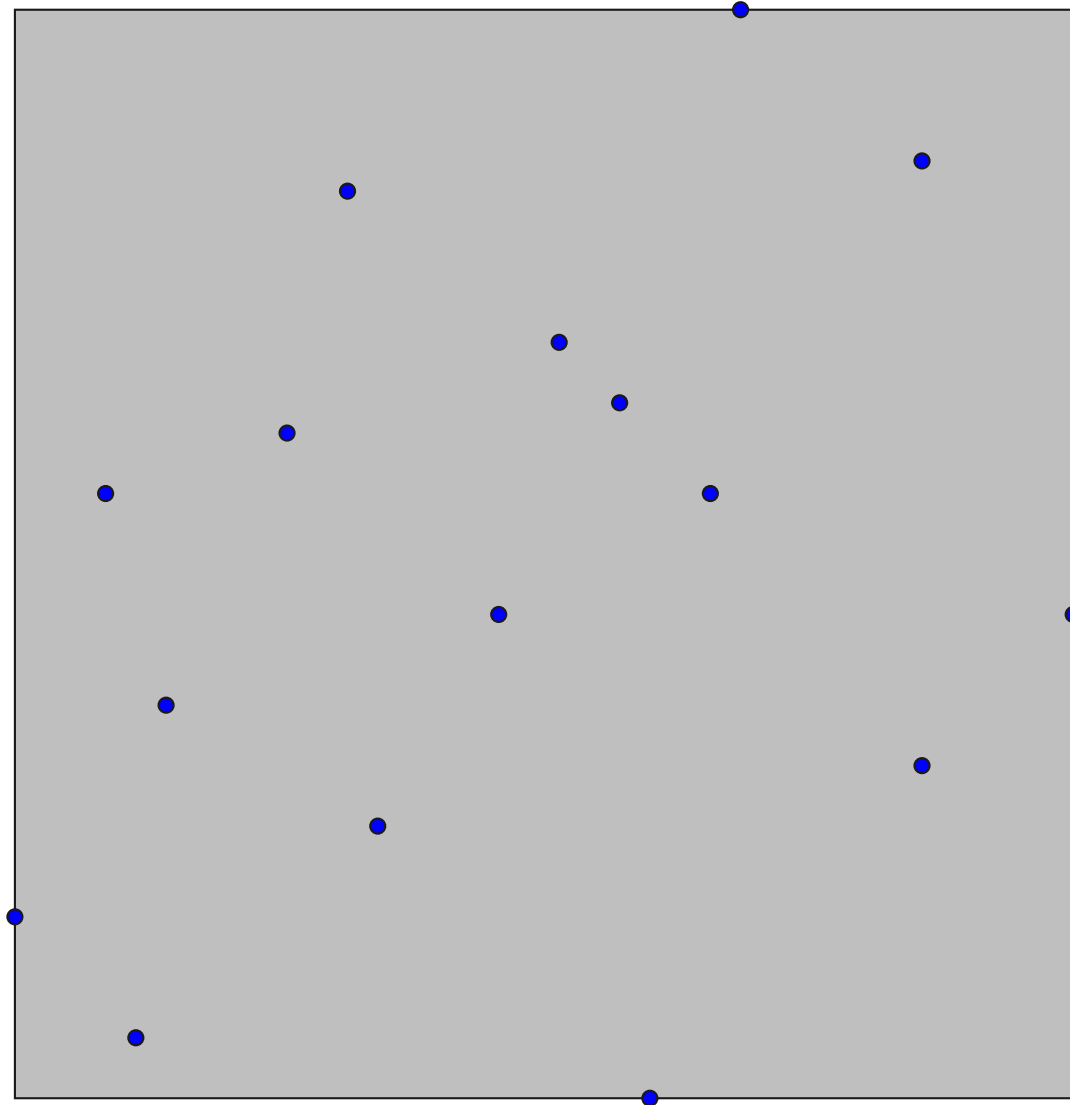
# The *kd*-tree



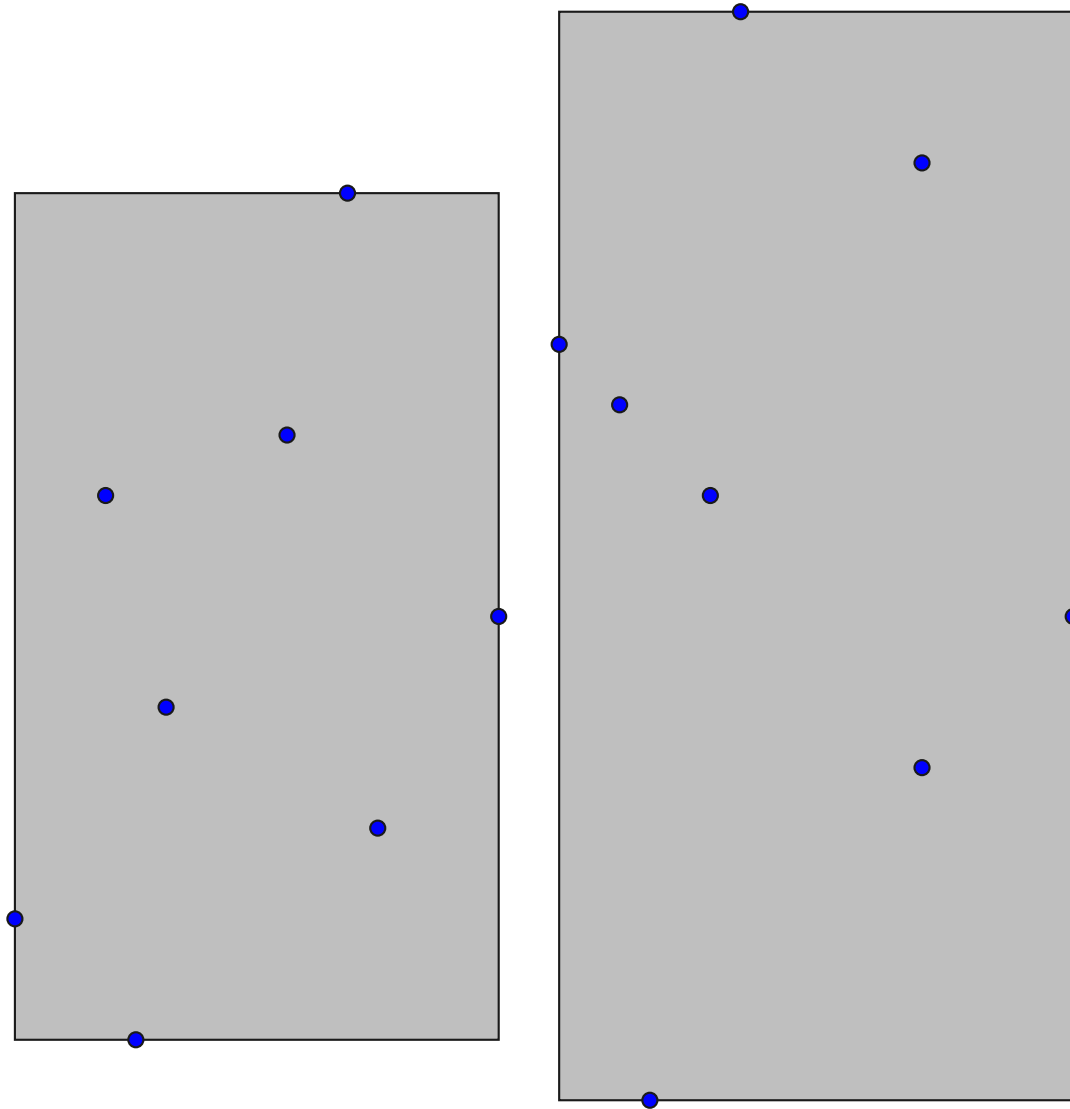
# The $kd$ -tree



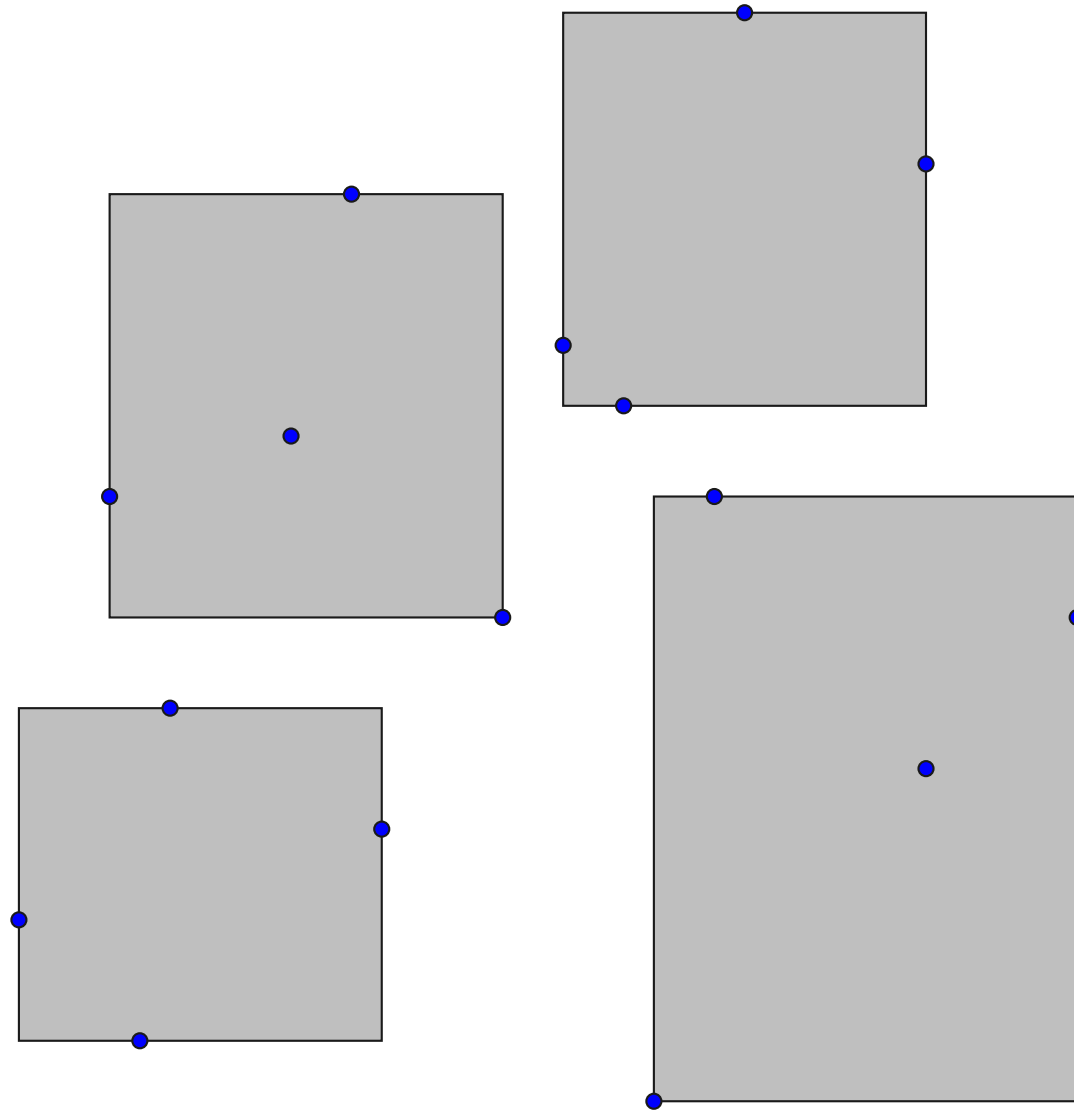
# The $kd$ -tree



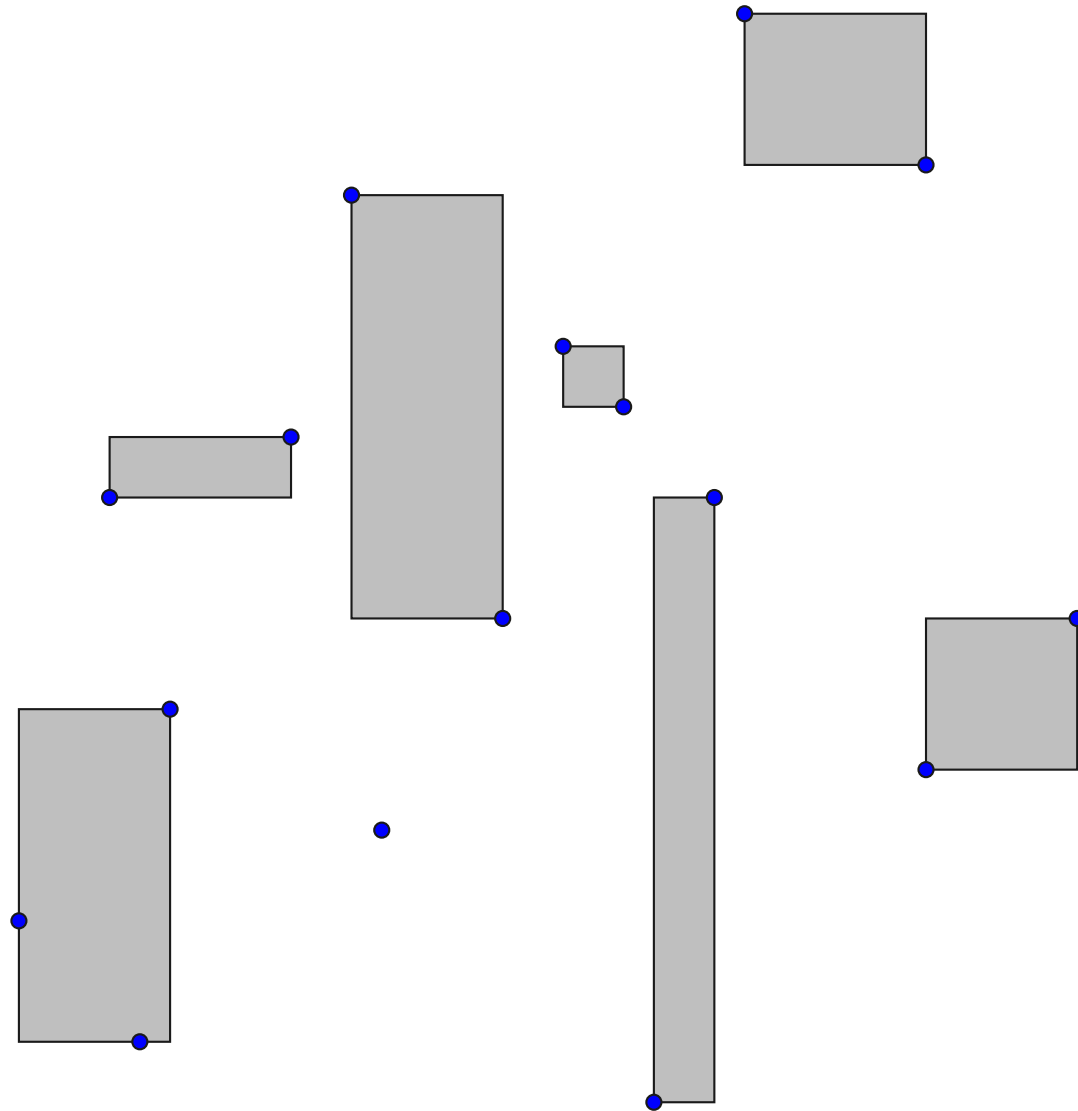
# The $kd$ -tree



# The $kd$ -tree

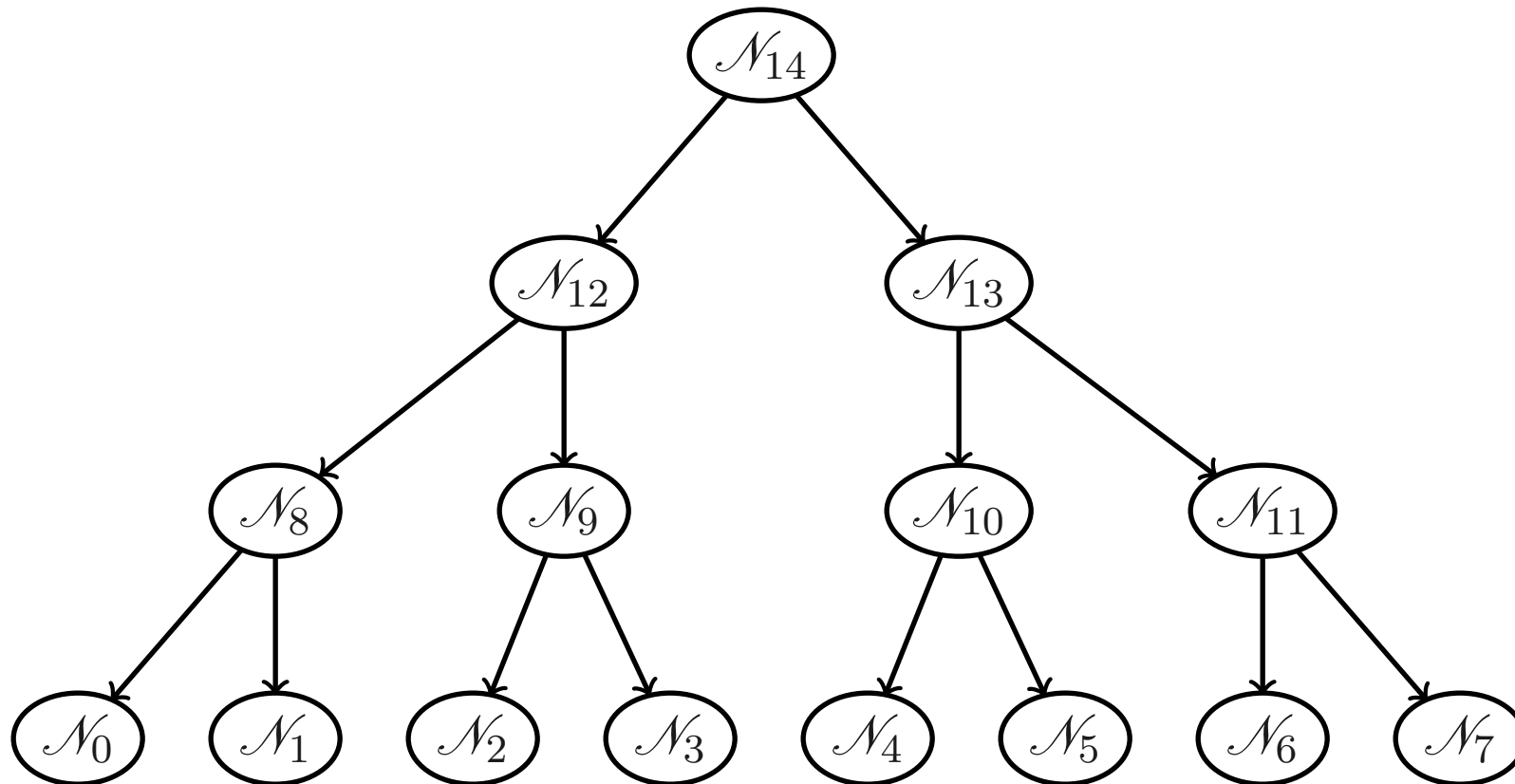


# The $kd$ -tree

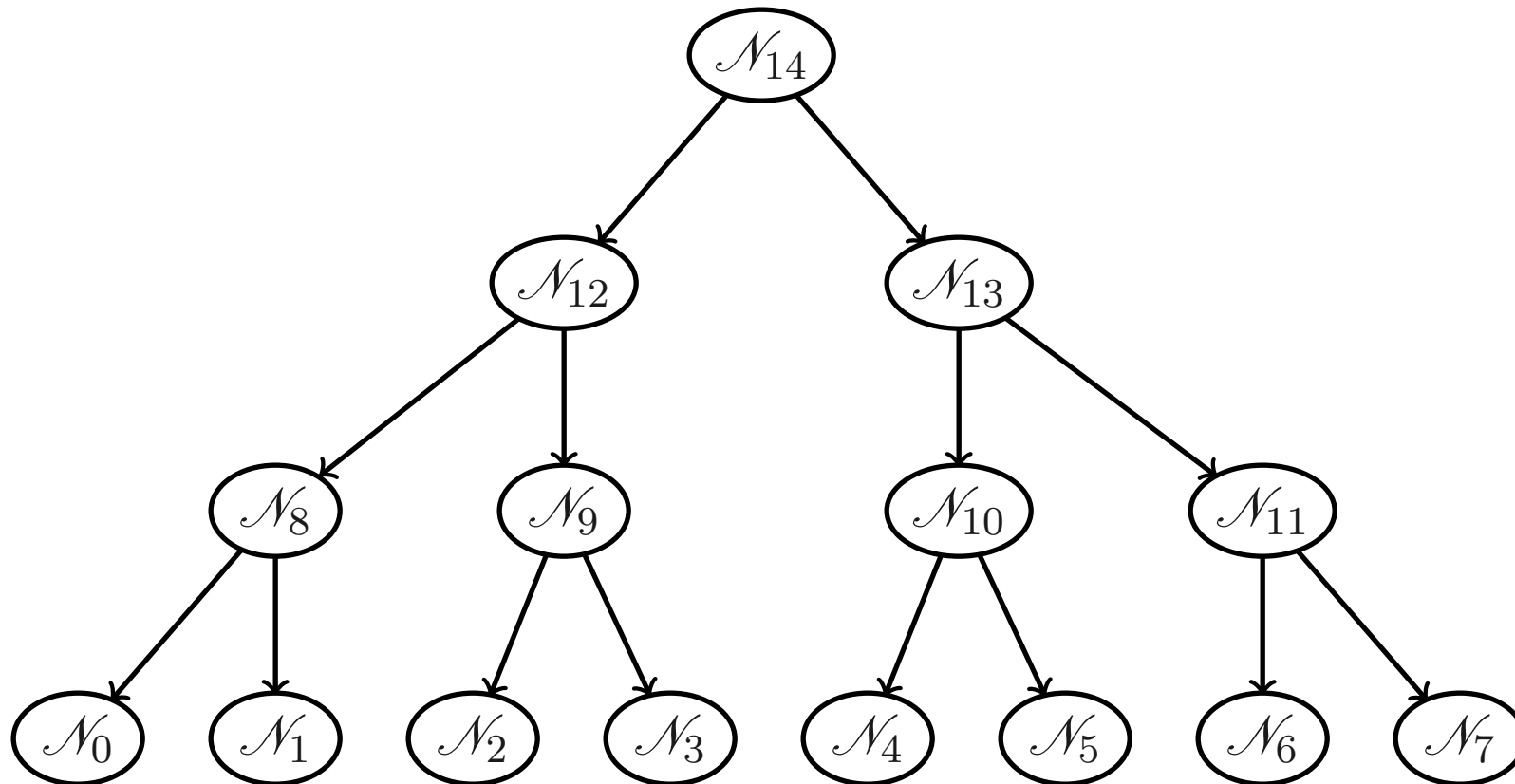




# The *kd*-tree



# The $kd$ -tree



A typical  $kd$ -tree only holds points in the leaves.

# *kd*-tree building algorithm

Written as a procedure:

1. Compute (minimum) bounding box of all points  $S$ . (This can be wrapped into subsequent steps...)

# *kd*-tree building algorithm

Written as a procedure:

1. Compute (minimum) bounding box of all points  $S$ . (This can be wrapped into subsequent steps...)
2. Select a dimension  $d$  to split on. (How about the dimension with maximum variance?)

# *kd*-tree building algorithm

Written as a procedure:

1. Compute (minimum) bounding box of all points  $S$ . (This can be wrapped into subsequent steps...)
2. Select a dimension  $d$  to split on. (How about the dimension with maximum variance?)
3. Select a value  $t$  to split on. (How about the mean in dimension  $d$ ? Or the median?)

# *kd*-tree building algorithm

Written as a procedure:

1. Compute (minimum) bounding box of all points  $S$ . (This can be wrapped into subsequent steps...)
2. Select a dimension  $d$  to split on. (How about the dimension with maximum variance?)
3. Select a value  $t$  to split on. (How about the mean in dimension  $d$ ? Or the median?)
4. Put all points with value less than  $t$  in dimension  $d$  into the left set  $S_l$ , and all others in the right set  $S_r$ . (Implementationally: one round of quicksort.)

# *kd*-tree building algorithm

Written as a procedure:

1. Compute (minimum) bounding box of all points  $S$ . (This can be wrapped into subsequent steps...)
2. Select a dimension  $d$  to split on. (How about the dimension with maximum variance?)
3. Select a value  $t$  to split on. (How about the mean in dimension  $d$ ? Or the median?)
4. Put all points with value less than  $t$  in dimension  $d$  into the left set  $S_l$ , and all others in the right set  $S_r$ . (Implementationally: one round of quicksort.)
5. Recurse with  $S_l$  and  $S_r$ ; the subtrees produced will be the left and right children of this node.

# *kd*-tree implementation notes

Trees **organize points hierarchically**, and each node corresponds to a **region of the input space**. But some hierarchies are better than others...



# *kd*-tree implementation notes

Trees **organize points hierarchically**, and each node corresponds to a **region of the input space**. But some hierarchies are better than others...

- Only hold points in the leaves. So nodes in the tree only hold a bounding box and two children, but no points.
- A leaf might hold something like 20 points. (For cache reasons...)
- Points held by a leaf should be contiguous in memory.
- Nodes should be as 'tight' as possible for pruning. (More soon...)
- Mean-split *kd*-trees tend to give better performance, at least for *k*-NN search.
- Splitting on the dimension with maximum variance tends to give better performance, at least for *k*-NN search.

# A diverse forest

R.A. Finkel and J.L. Bentley. “Quad trees: a data structure for retrieval on composite keys,” *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.

J.L. Bentley. “Multidimensional binary search trees used for associative searching,” *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

K. Fukunaga and P.M. Narendra. “A branch and bound algorithm for computing k-nearest neighbors,” *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 750–753, 1975.

C.L. Jackins and S.L. Tanimoto. “Oct-trees and their use in representing three-dimensional objects,” *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.

J.K. Uhlmann. “Satisfying general proximity/similarity queries with metric trees,” *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991.

P.N. Yianilos. “Data structures and algorithms for nearest neighbor search in general metric spaces,” in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pp. 311–321, 1993.

A.W. Moore. “Very fast EM-based mixture model clustering using multiresolution kd-trees,” in *Advances in Neural Information Processing Systems 11 (NIPS '98)*, pp. 543–549, 1999.

J. McNames. “A fast nearest-neighbor algorithm based on a principal axis search tree,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 9, pp. 964–976, 2001.

O. Procopiuc, P.K. Agarwal, L. Arge, and J.S. Vitter. “Bkd-tree: a dynamic scalable kd-tree,” in *Advances in Spatial and Temporal Databases*, pp. 46–65, 2003.

A. Beygelzimer, S.M. Kakade, and J. Langford. “Cover trees for nearest neighbor,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–106, 2006.

S. Dasgupta and Y. Freund. “Random projection trees and low dimensional manifolds,” in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC '08)*, pp. 537–546, 2008.

# Space trees

Here's a unifying abstraction for the types of trees we care about in  $\mathcal{U}_{\text{temp}}$ .

A **space tree** on a dataset  $S \in \mathfrak{R}^{n \times d}$ , is an undirected, acyclic, rooted simple graph with the following properties:

# Space trees

Here's a unifying abstraction for the types of trees we care about in  $\mathcal{U}_{\text{temp}}$ .

A **space tree** on a dataset  $S \in \mathbb{R}^{n \times d}$ , is an undirected, acyclic, rooted simple graph with the following properties:

- Each node (or vertex) holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).

# Space trees

Here's a unifying abstraction for the types of trees we care about in  $\mathcal{U}_{\text{temp}}$ .

A **space tree** on a dataset  $S \in \mathfrak{R}^{n \times d}$ , is an undirected, acyclic, rooted simple graph with the following properties:

- Each node (or vertex) holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).
- There is one node in every space tree with no parent; this is the root of the tree.

# Space trees

Here's a unifying abstraction for the types of trees we care about in  $\mathcal{U}_{\text{temp}}$ .

A **space tree** on a dataset  $S \in \mathbb{R}^{n \times d}$ , is an undirected, acyclic, rooted simple graph with the following properties:

- Each node (or vertex) holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).
- There is one node in every space tree with no parent; this is the root of the tree.
- Each point in  $S$  is contained in at least one node of the tree.

# Space trees

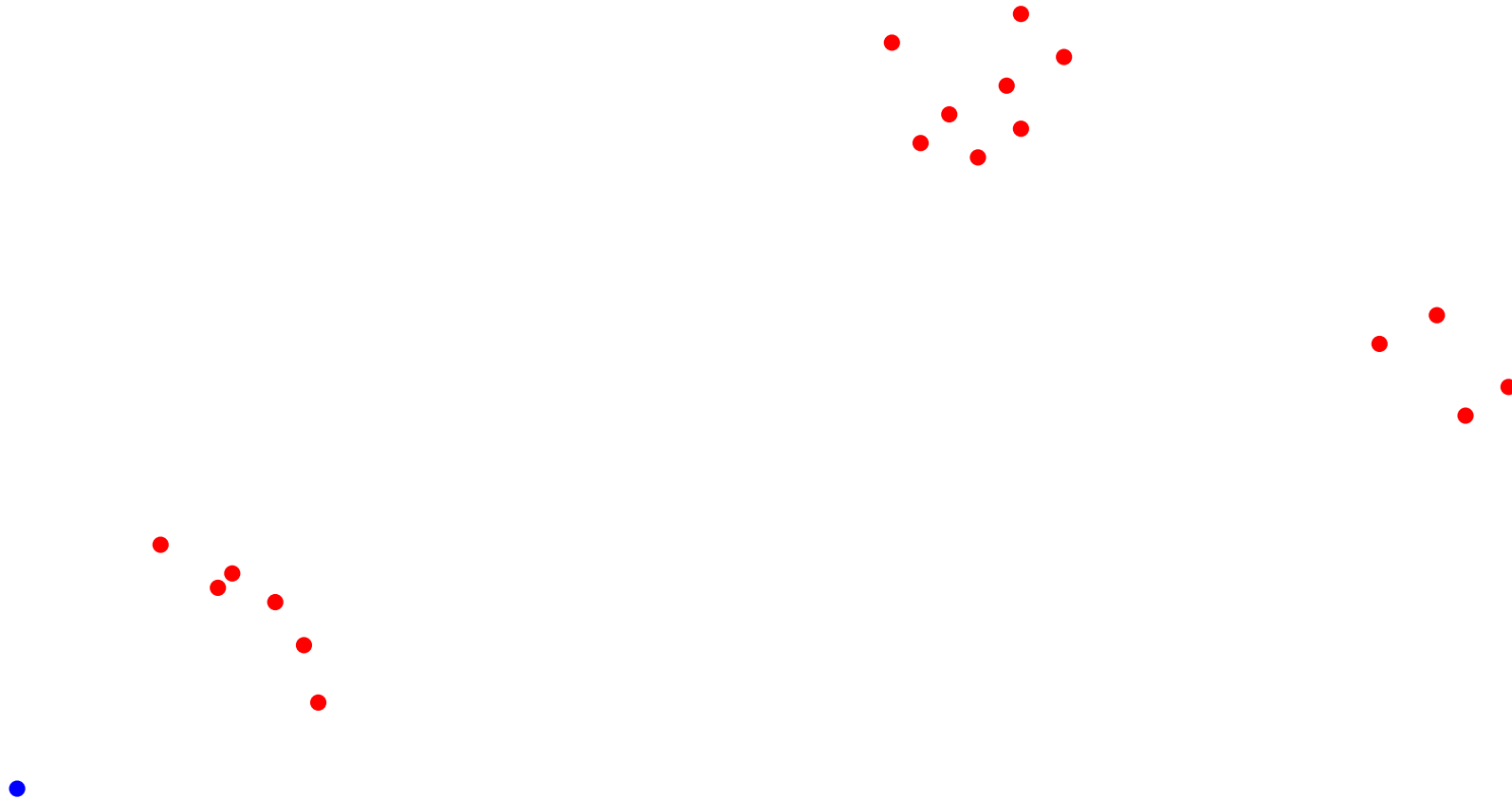
Here's a unifying abstraction for the types of trees we care about in  $\mathcal{U}_{\text{temp}}$ .

A **space tree** on a dataset  $S \in \mathbb{R}^{n \times d}$ , is an undirected, acyclic, rooted simple graph with the following properties:

- Each node (or vertex) holds a number of points (possibly zero) and is connected to one parent node and a number of child nodes (possibly zero).
- There is one node in every space tree with no parent; this is the root of the tree.
- Each point in  $S$  is contained in at least one node of the tree.
- Each node  $\mathcal{N}$  of the tree has a convex subset of  $\mathbb{R}^d$  that contains each of the points in that node as well as the convex subsets represented by each child of the node.

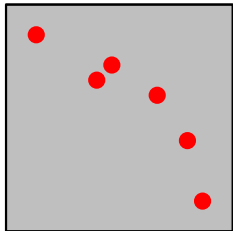
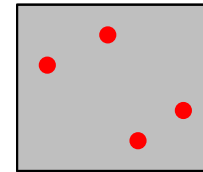
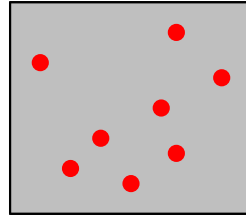
R.R. Curtin, W.B. March, P. Ram, D.V. Anderson, A.G. Gray, C.L. Isbell, Jr. "Tree-independent dual-tree algorithms," in *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, pp. 1435–1443, Atlanta, GA, 2013.

# A Geometric Observation

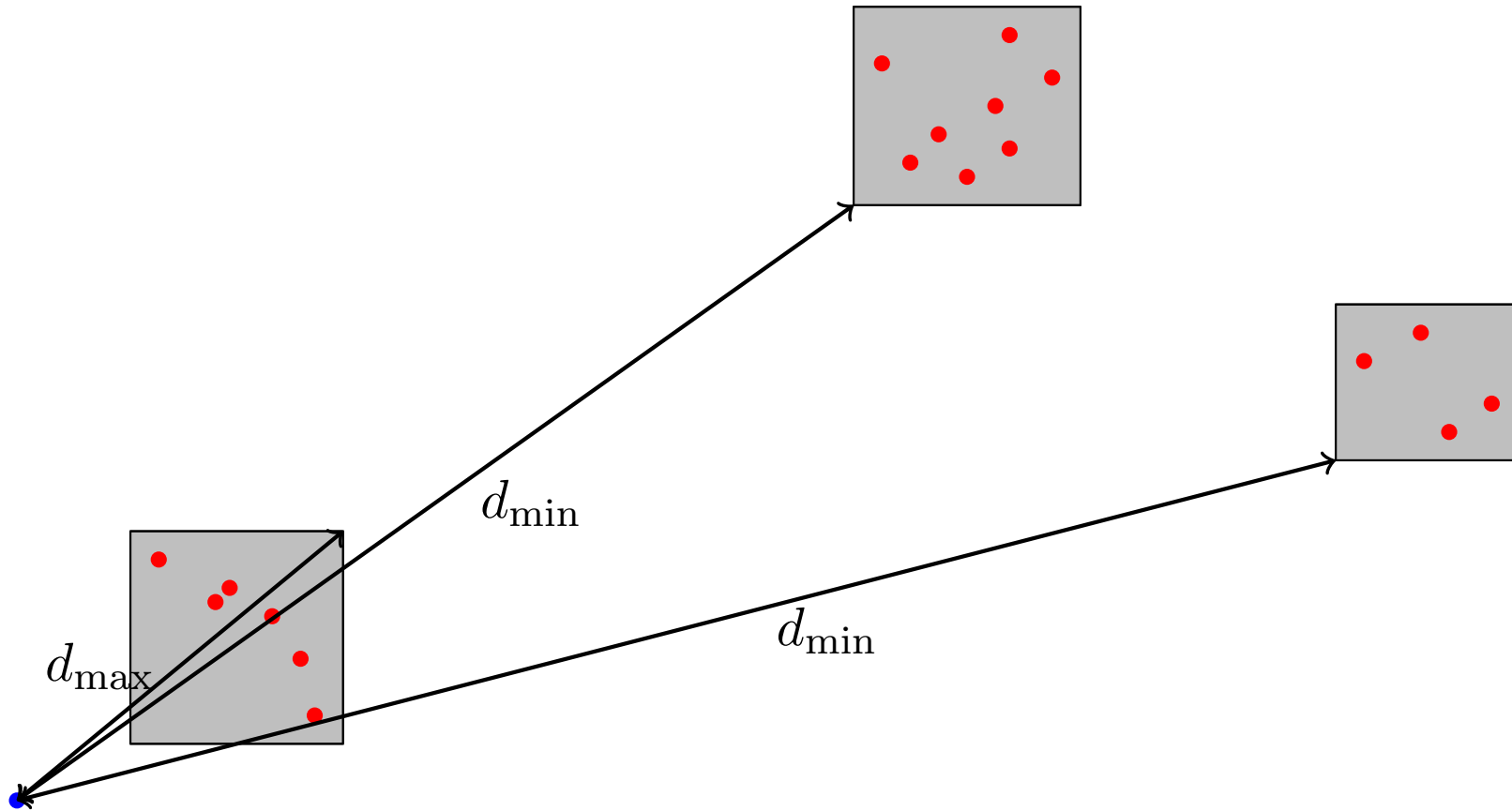




# A Geometric Observation (2)



# A Geometric Observation (3)



# Return to the universe

How do we use a  $kd$ -tree to solve the range search query? Assume we built a tree  $\mathcal{T}_r$  on the reference set  $S_r$ .

# Return to the universe

How do we use a  $kd$ -tree to solve the range search query? Assume we built a tree  $\mathcal{T}_r$  on the reference set  $S_r$ .

```
Recurse( $p_q, [l, u], \mathcal{N}_i$ ):  
  if  $d_{\max}(p_q, \mathcal{N}_i) < l$  or  $d_{\min}(p_q, \mathcal{N}_i) > u$ :  
    return;  
  
  for (point  $p_r : \mathcal{N}_i$ ):  
    BaseCase( $p_q, p_r, [l, u]$ )  
  
  for (child  $\mathcal{N}_c : \mathcal{N}_i$ ):  
    Recurse( $p_q, [l, u], \mathcal{N}_c$ )
```

# Return to the universe

How do we use a  $kd$ -tree to solve the range search query? Assume we built a tree  $\mathcal{T}_r$  on the reference set  $S_r$ .

Recurse( $p_q, [l, u], \mathcal{N}_i$ ):

if  $d_{\max}(p_q, \mathcal{N}_i) < l$  or  $d_{\min}(p_q, \mathcal{N}_i) > u$ :  
return;

for (point  $p_r$  :  $\mathcal{N}_i$ ):  
BaseCase( $p_q, p_r, [l, u]$ )

for (child  $\mathcal{N}_c$ :  $\mathcal{N}_i$ ):  
Recurse( $p_q, [l, u], \mathcal{N}_c$ )

BaseCase( $p_q, p_r, [l, u]$ ):

if ( $d(p_q, p_r) \in [l, u]$ ):  
add  $p_r$  to result set

# Return to the universe

How do we use a  $kd$ -tree to solve the range search query? Assume we built a tree  $\mathcal{T}_r$  on the reference set  $S_r$ .

```
Recurse( $p_q, [l, u], \mathcal{N}_i$ ):
```

```
  if  $d_{\max}(p_q, \mathcal{N}_i) < l$  or  $d_{\min}(p_q, \mathcal{N}_i) > u$ :  
    return;
```

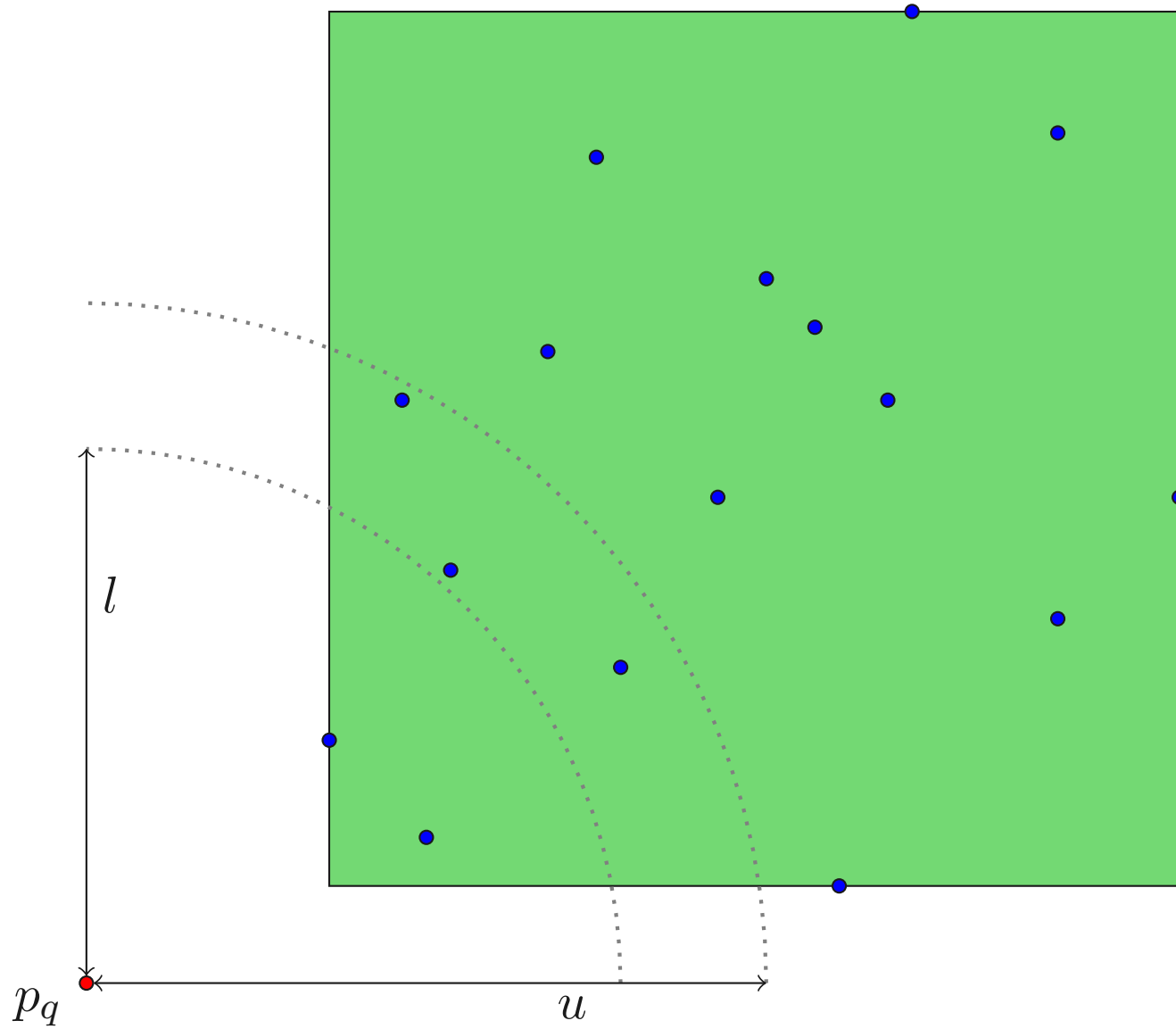
```
  for (point  $p_r : \mathcal{N}_i$ ):  
    BaseCase( $p_q, p_r, [l, u]$ )
```

```
  for (child  $\mathcal{N}_c : \mathcal{N}_i$ ):  
    Recurse( $p_q, [l, u], \mathcal{N}_c$ )
```

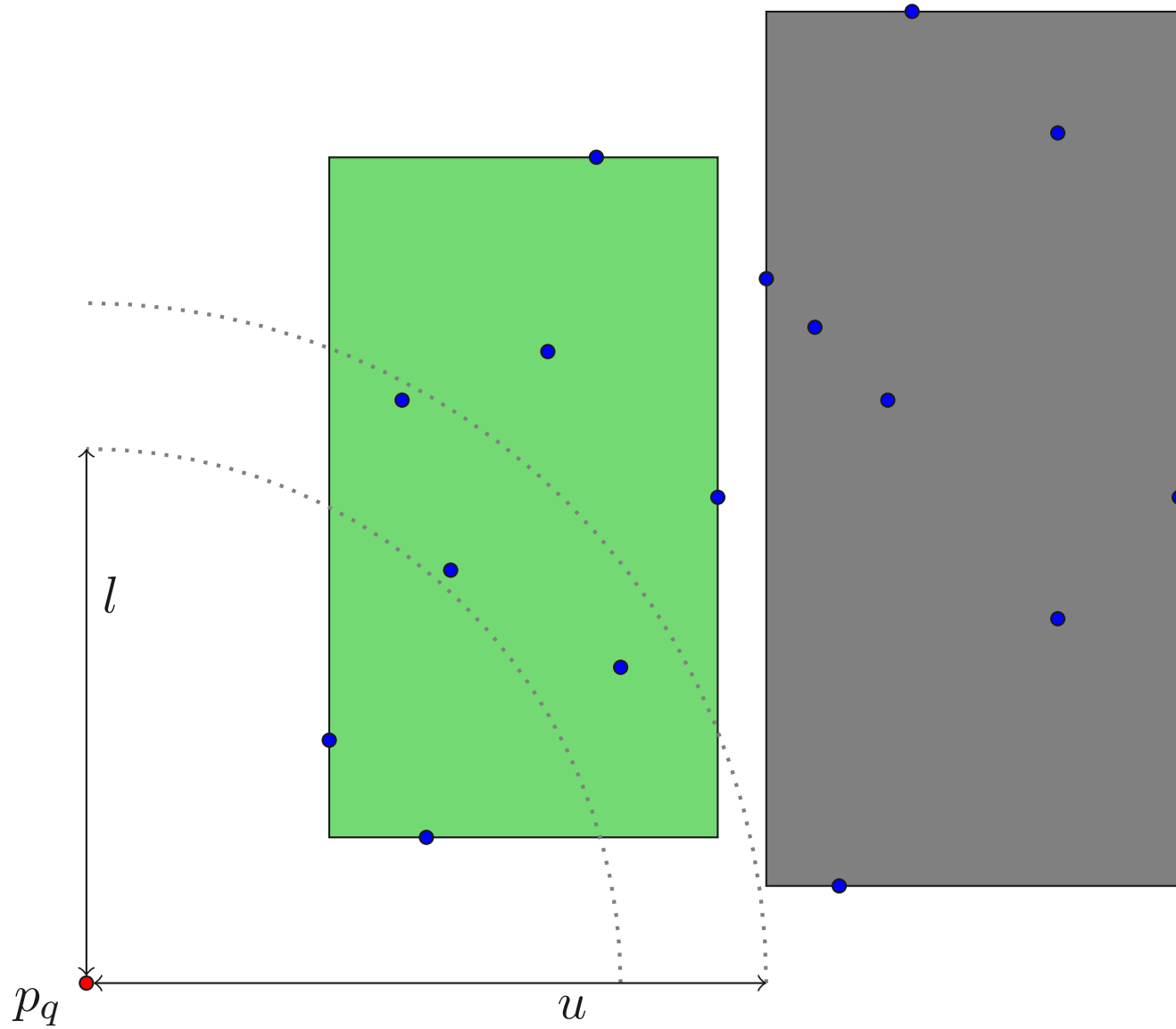
```
BaseCase( $p_q, p_r, [l, u]$ ):  
  if ( $d(p_q, p_r) \in [l, u]$ )  
    add  $p_r$ 
```

Depending on the dataset, this approach can be very fast compared to brute-force search!

# Range search

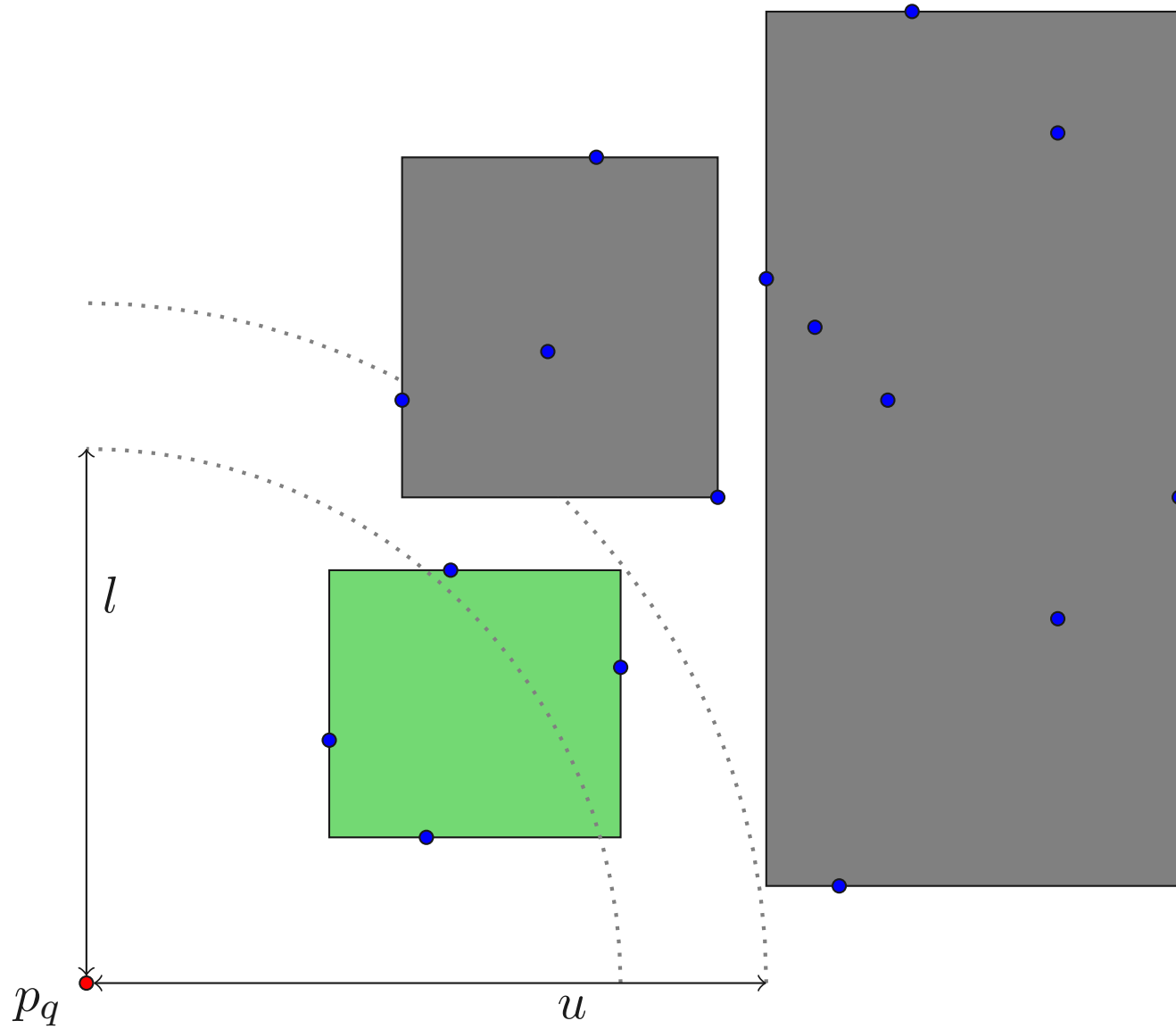


# Range search

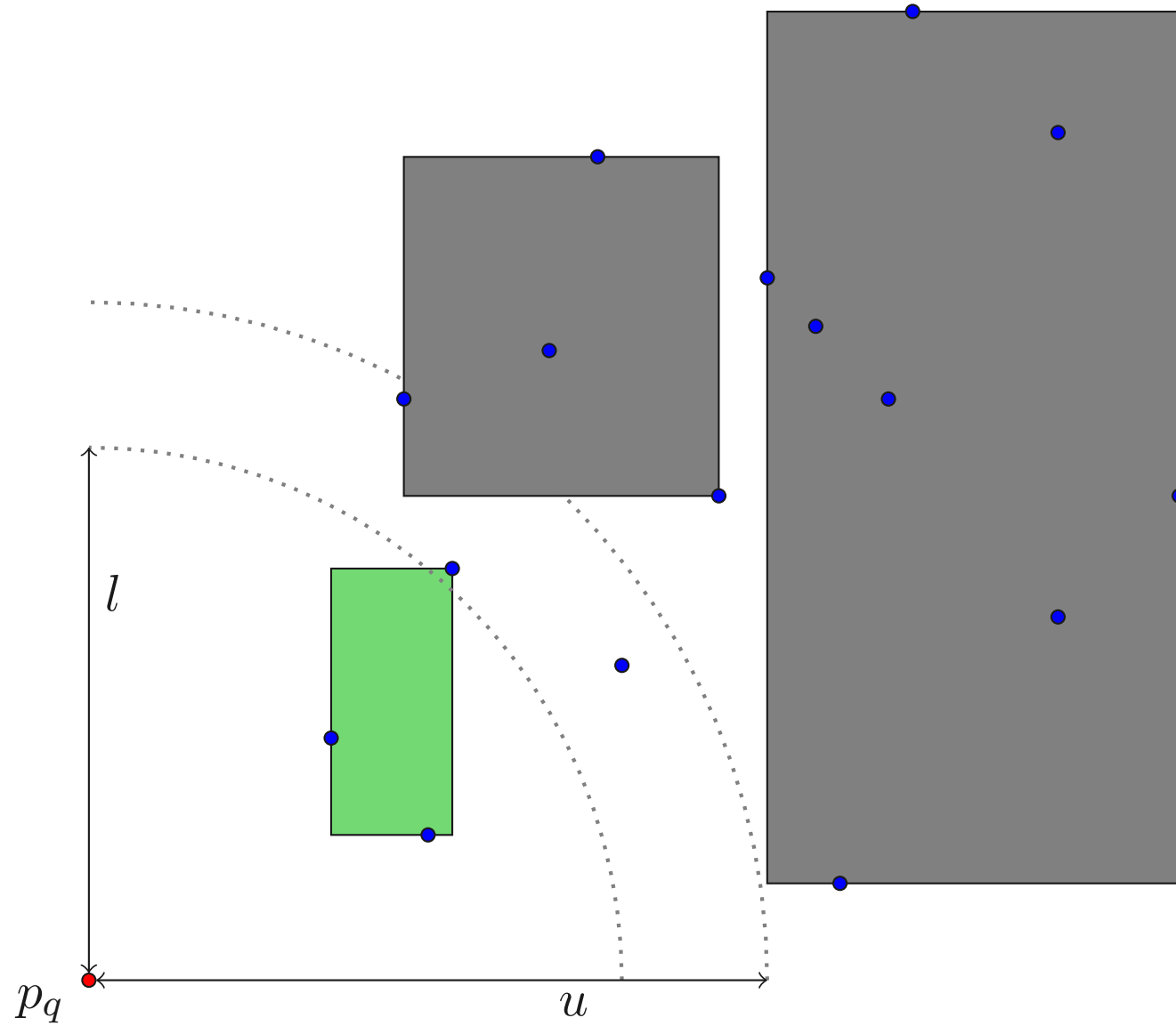




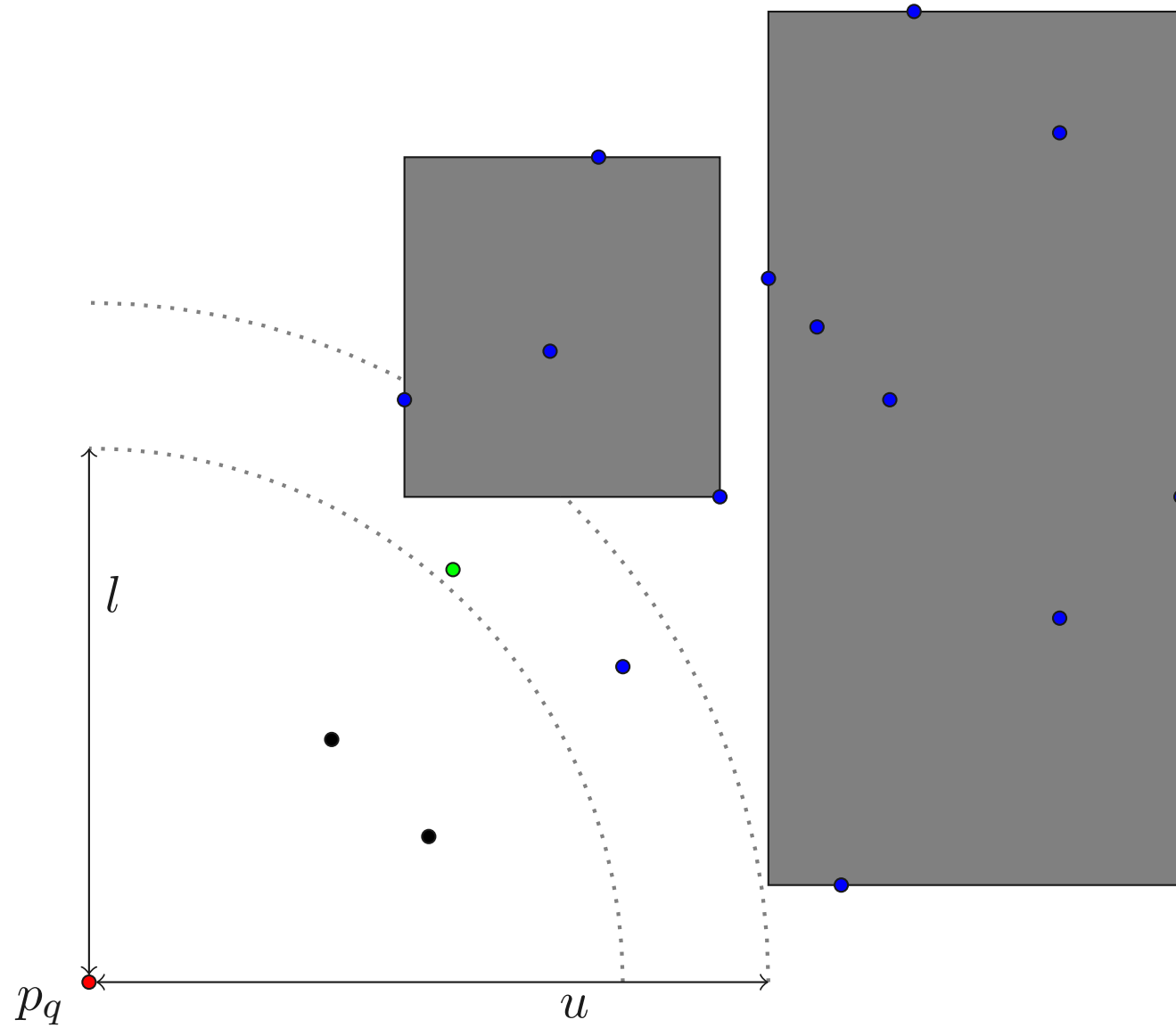
# Range search



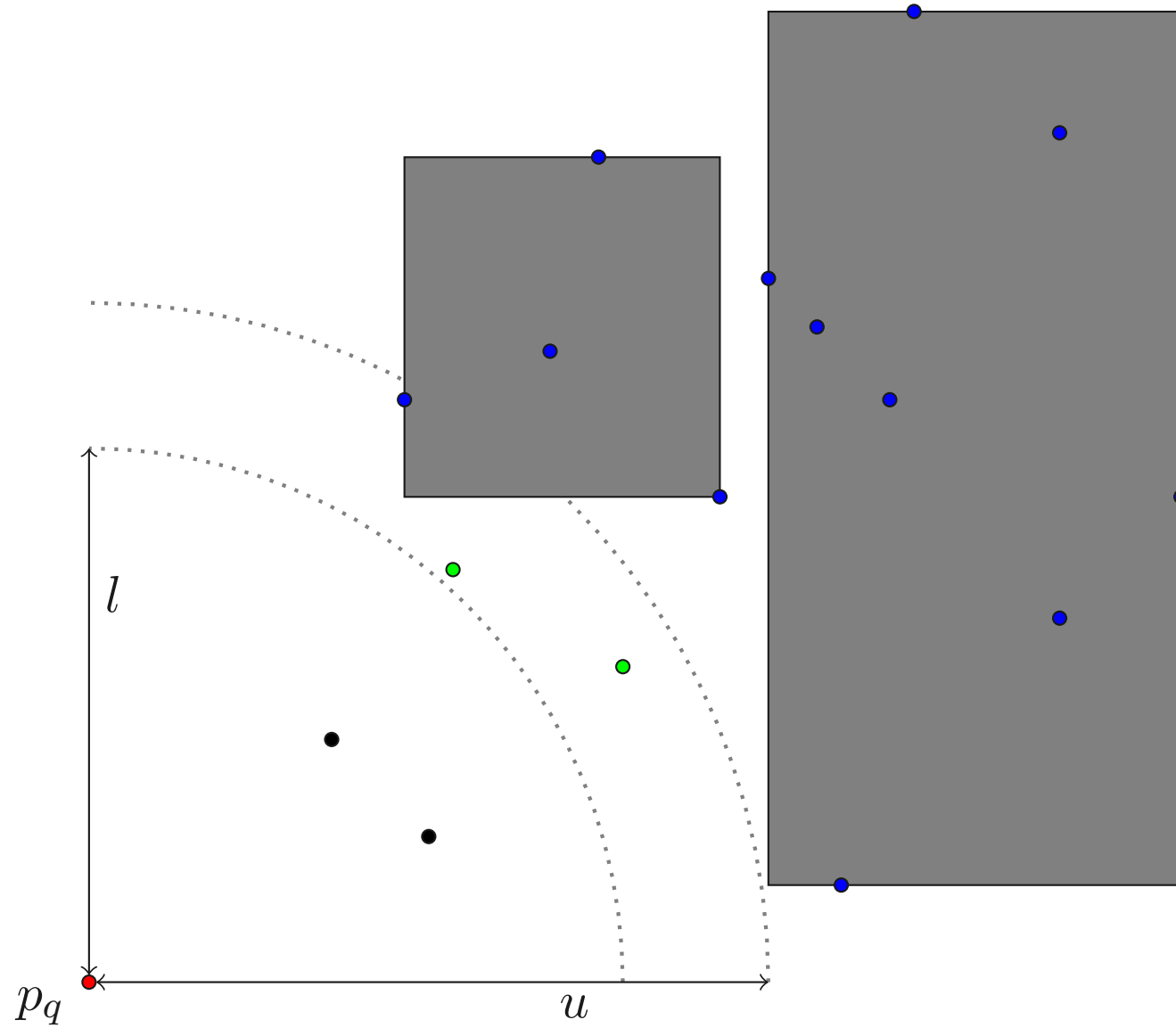
# Range search



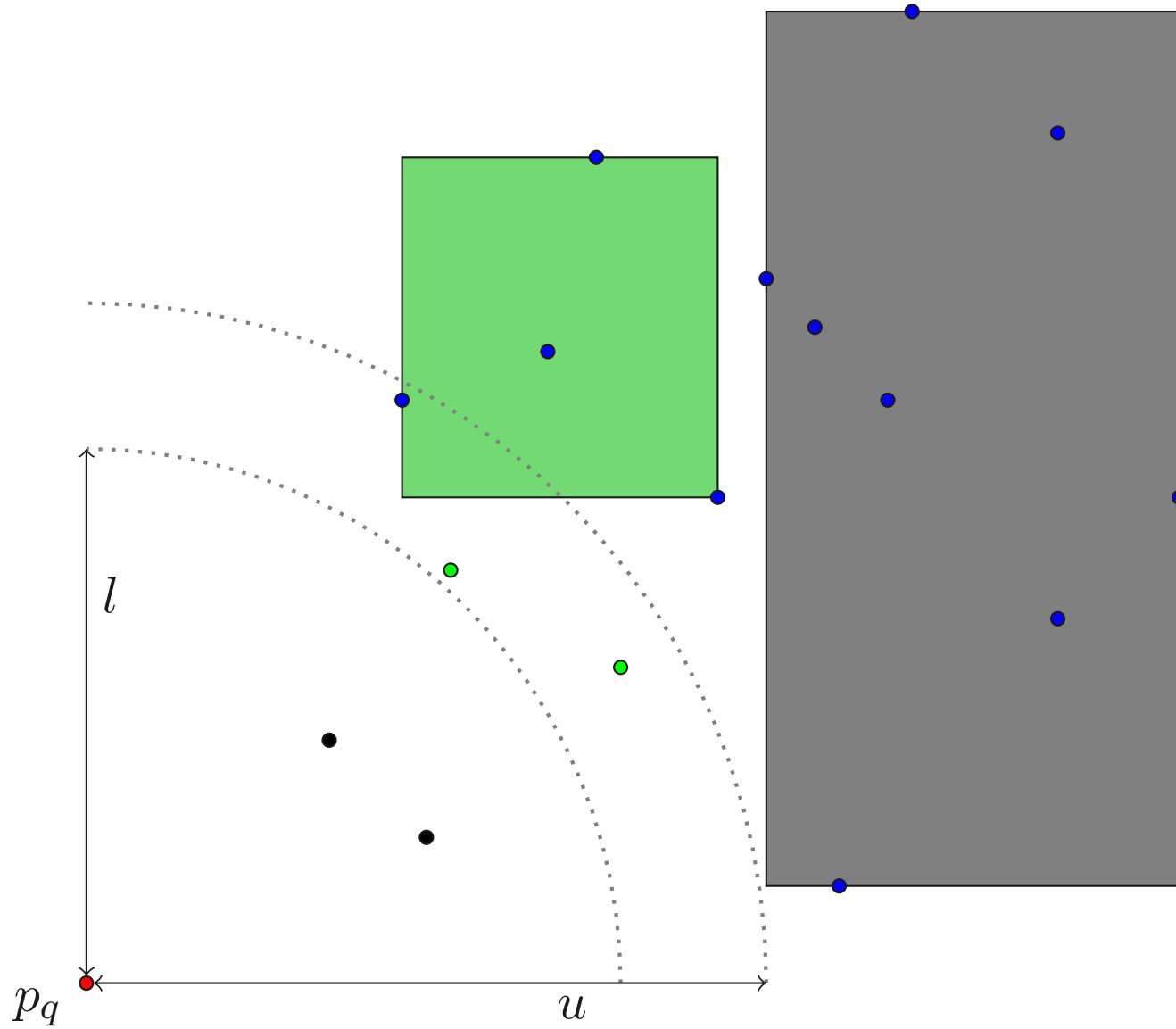
# Range search



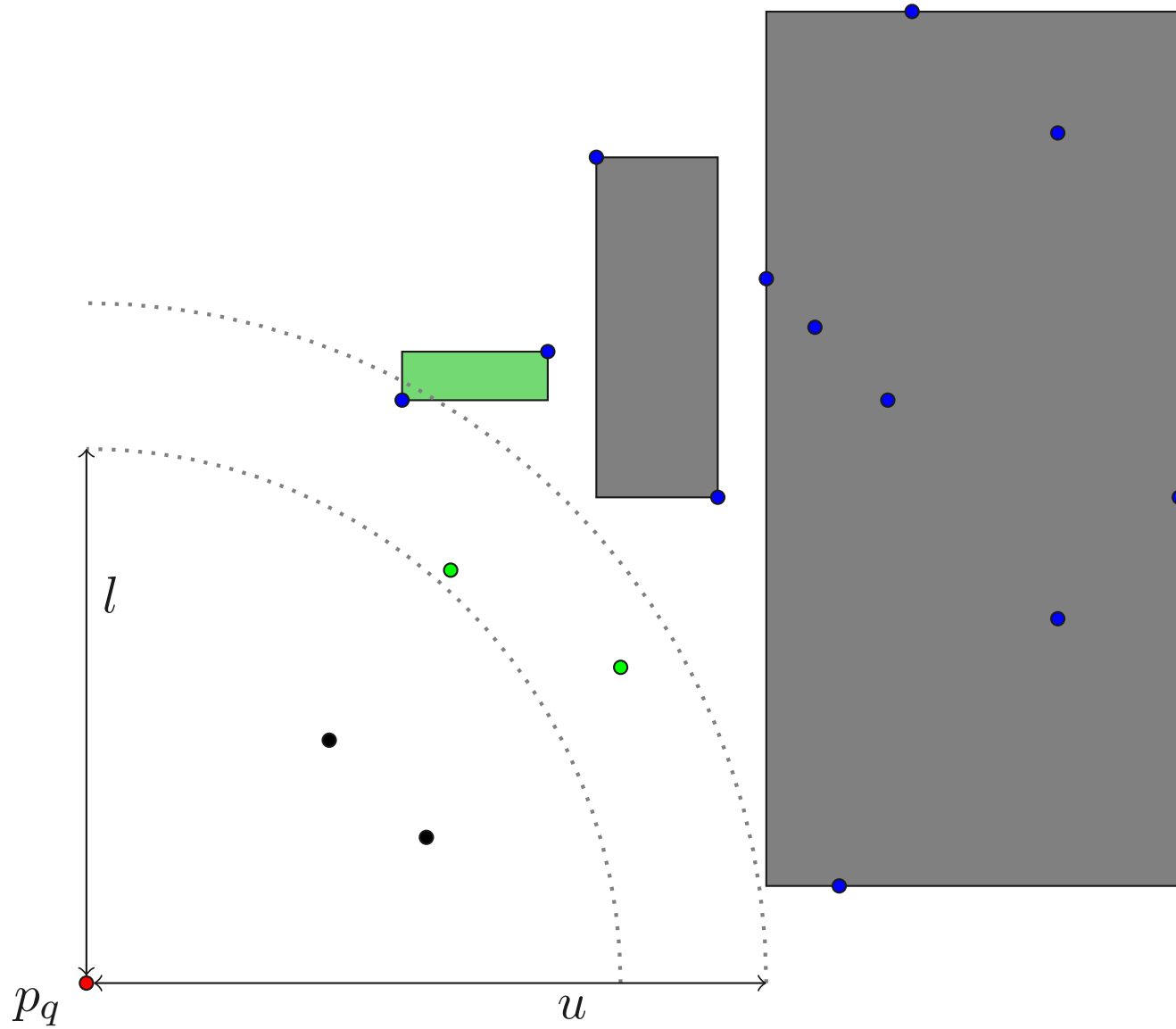
# Range search



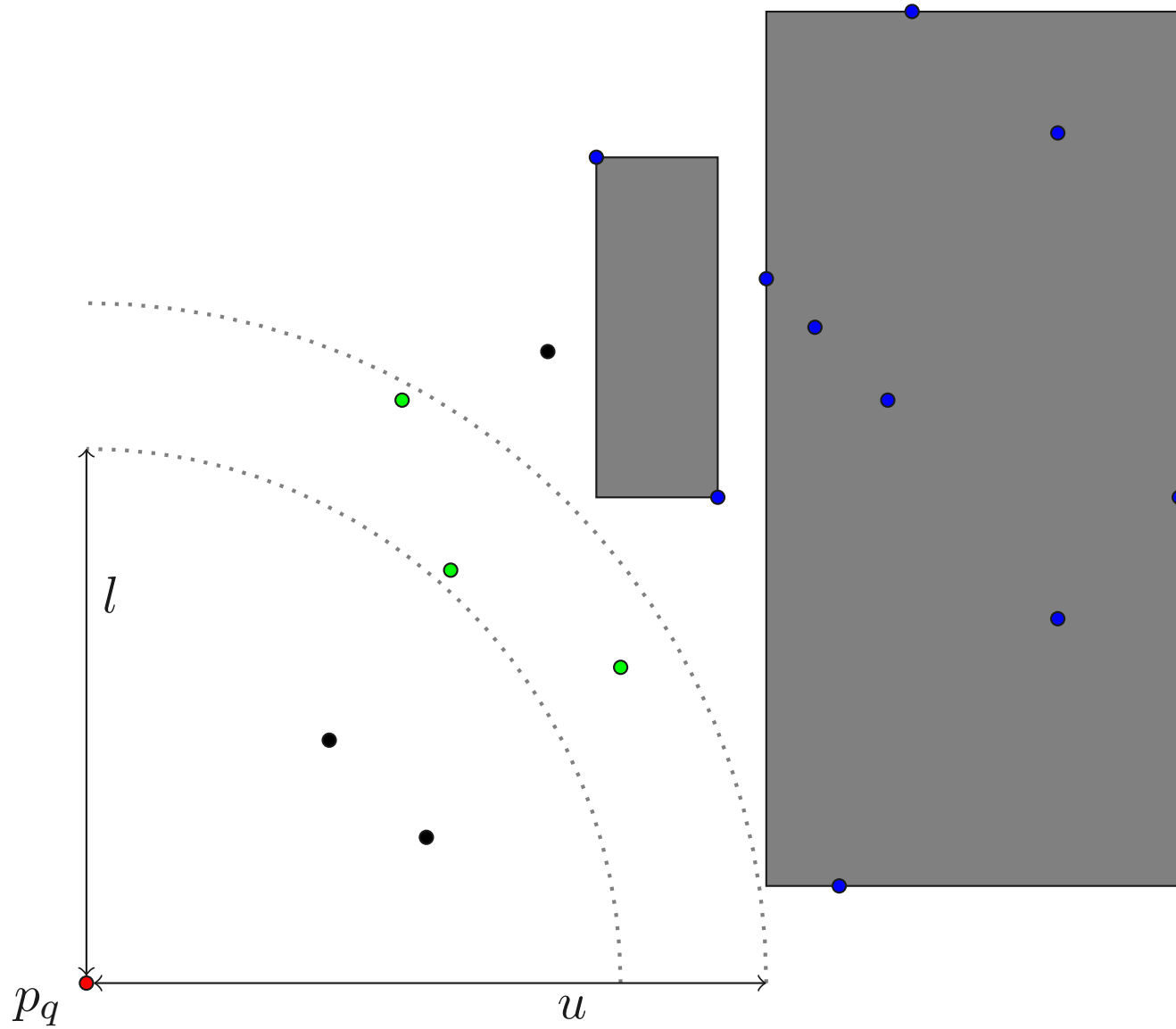
# Range search



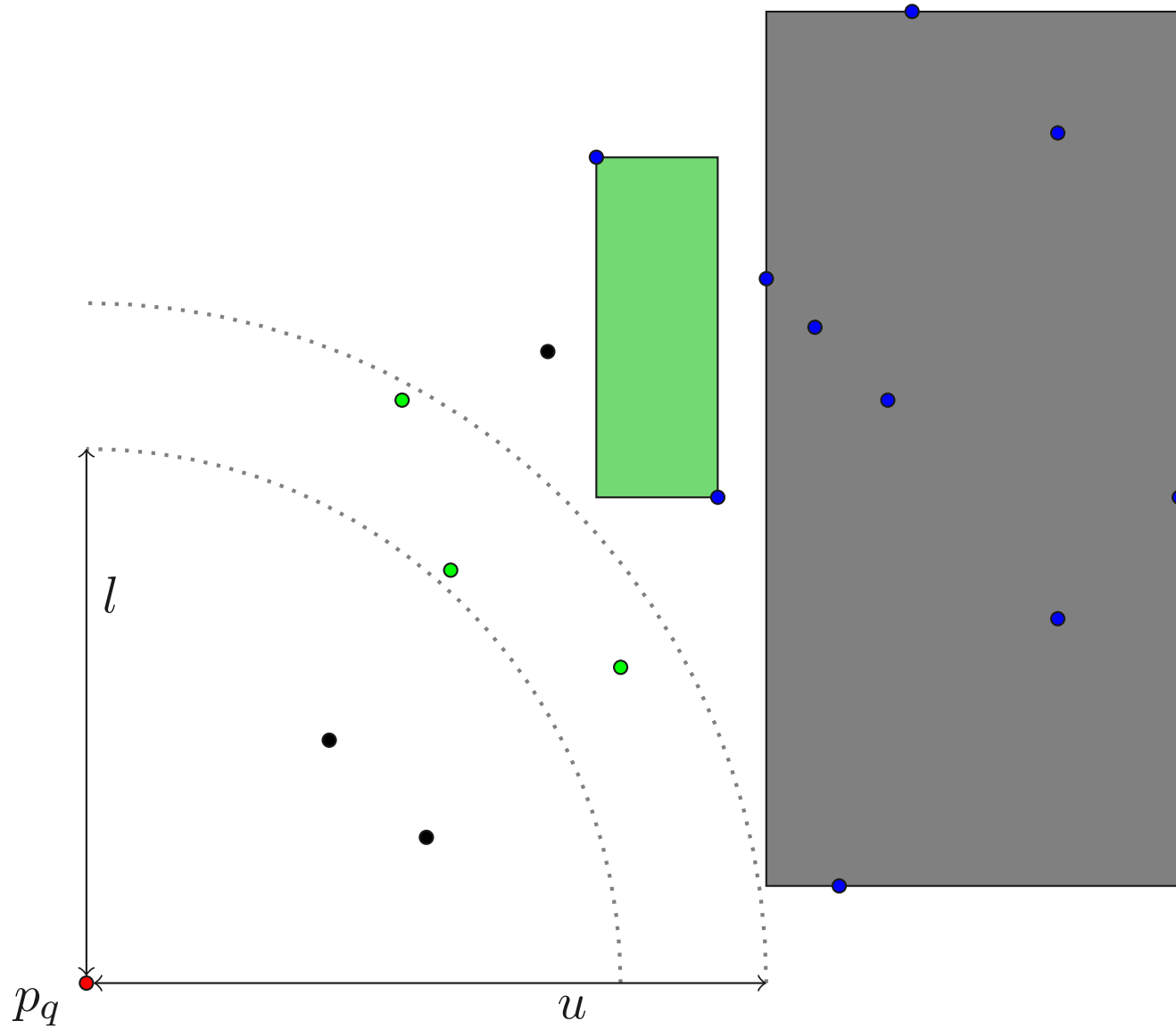
# Range search



# Range search

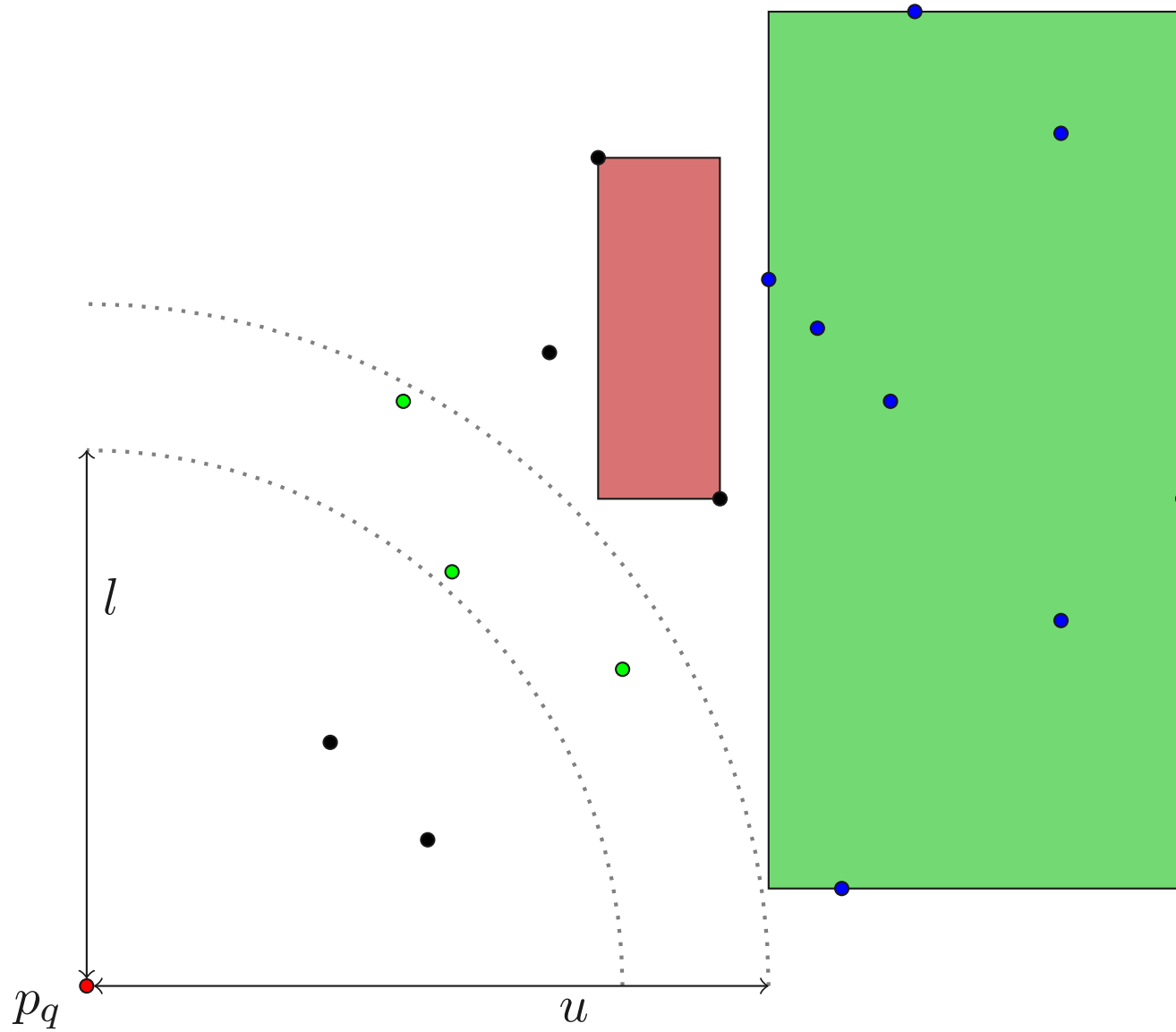


# Range search

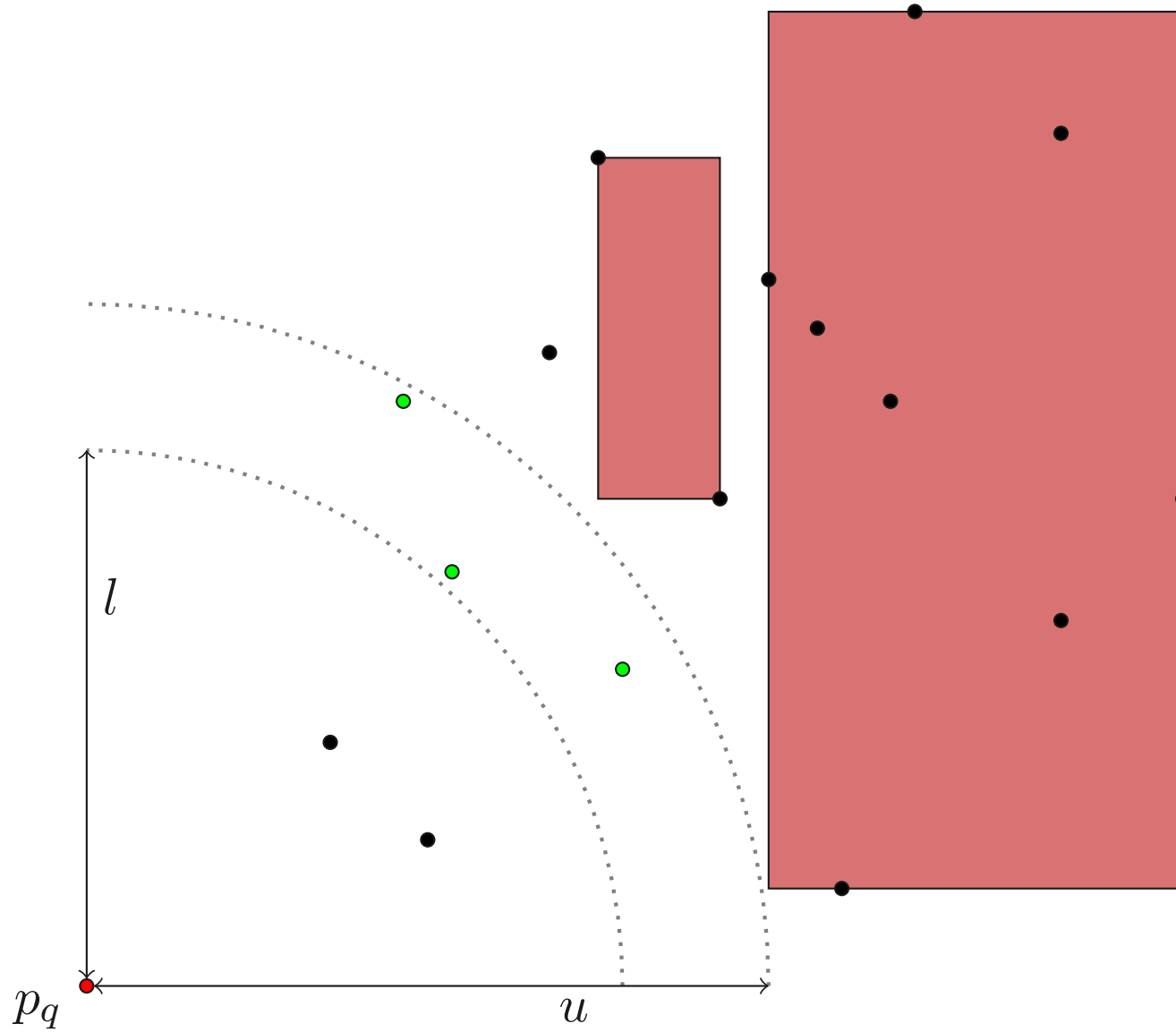




# Range search



# Range search



# 1-nearest-neighbor Search

Generalizes to  $k$ -NN with a little bit more overhead...

```
Recurse( $p_q, u, \mathcal{N}_i$ ):  
  if  $d_{\min}(p_q, \mathcal{N}_i) > u$ :  
    return;  
  
  for (point  $p_r : \mathcal{N}_i$ ):  
     $u \leftarrow \text{BaseCase}(p_q, p_r, u)$   
  
  for (child  $\mathcal{N}_c : \mathcal{N}_i$ ):  
    Recurse( $p_q, u, \mathcal{N}_c$ )
```

# 1-nearest-neighbor Search

Generalizes to  $k$ -NN with a little bit more overhead...

Recurse( $p_q, u, \mathcal{N}_i$ ):

  if  $d_{\min}(p_q, \mathcal{N}_i) > u$ :  
    return;

  for (point  $p_r : \mathcal{N}_i$ ):  
     $u \leftarrow \text{BaseCase}(p_q, p_r, u)$

  for (child  $\mathcal{N}_c : \mathcal{N}_i$ ):  
    Recurse( $p_q, u, \mathcal{N}_c$ )

BaseCase( $p_q, p_r, u$ ):

  if ( $d(p_q, p_r) < u$ ):  
     $p_r$  is the new nearest neighbor candidate  
    return  $d(p_q, p_r)$

return  $u$

# Across the Universe

- R.A. Finkel and J.L. Bentley. "Quad trees: a data structure for retrieval on composite keys," *Acta Informatica*, vol. 4, no. 1, pp. 1–9, 1974.
- J.L. Bentley. "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.
- K. Fukunaga and P.M. Narendra. "A branch and bound algorithm for computing k-nearest neighbors," *IEEE Transactions on Computers*, vol. 100, no. 7, pp. 750–753, 1975.
- C.L. Jackins and S.L. Tanimoto. "Oct-trees and their use in representing three-dimensional objects," *Computer Graphics and Image Processing*, vol. 14, no. 3, pp. 249–270, 1980.
- J.K. Uhlmann. "Satisfying general proximity/similarity queries with metric trees," *Information Processing Letters*, vol. 40, no. 4, pp. 175–179, 1991.
- P.N. Yianilos. "Data structures and algorithms for nearest neighbor search in general metric spaces," in *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA '93)*, pp. 311–321, 1993.
- A.W. Moore. "Very fast EM-based mixture model clustering using multiresolution kd-trees," in *Advances in Neural Information Processing Systems 11 (NIPS '98)*, pp. 543–549, 1999.
- J. McNames. "A fast nearest-neighbor algorithm based on a principal axis search tree," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 23, no. 9, pp. 964–976, 2001.
- O. Procopiuc, P.K. Agarwal, L. Arge, and J.S. Vitter. "Bkd-tree: a dynamic scalable kd-tree," in *Advances in Spatial and Temporal Databases*, pp. 46–65, 2003.
- A. Beygelzimer, S.M. Kakade, and J. Langford. "Cover trees for nearest neighbor," in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–106, 2006.
- S. Dasgupta and Y. Freund. "Random projection trees and low dimensional manifolds," in *Proceedings of the 40th Annual ACM Symposium on Theory of Computing (STOC '08)*, pp. 537–546, 2008.

# Intermediate question break

(we're at about 1979 by the way)

# Large query sets

For many problems (including our  $\mathcal{U}_{\text{temp}}$ ), we want results for more than just one query point!

- **all-nearest-neighbors**: for every point in  $S_r$ , tell me its nearest neighbor
- **batch nearest neighbor**: for a query set  $S_q$ , find the nearest neighbor of each  $p_q \in S_q$  in  $S_r$
- **all-range-search** and **batch range search** exist too...

Does this put us in  $\mathcal{U}'_{\text{temp}}$ ? That's up to you to decide.

# Dual-tree algorithms

I just described a **single-tree algorithm**. In that, we reduced computation by *bounding results for many points in  $S_r$  at once*.



# Dual-tree algorithms

I just described a **single-tree algorithm**. In that, we reduced computation by *bounding results for many points in  $S_r$  at once*.

In a **dual-tree algorithm**, we're going to also bound results for *many points in  $S_q$  at once*.

# What is a dual-tree algorithm?

We build two trees ( $\mathcal{T}_q$  and  $\mathcal{T}_r$ ).

# What is a dual-tree algorithm?

We build two trees ( $\mathcal{T}_q$  and  $\mathcal{T}_r$ ).

We'll traverse the trees simultaneously. (This may be dual breadth-first, dual depth-first, or some combination thereof.)

# What is a dual-tree algorithm?

We build two trees ( $\mathcal{I}_q$  and  $\mathcal{I}_r$ ).

We'll traverse the trees simultaneously. (This may be dual breadth-first, dual depth-first, or some combination thereof.)

When we visit a node pair (query and reference nodes), we'll see if we can prune that pair. Otherwise, we'll perform some base case between points contained in the two nodes.

# What is a dual-tree algorithm?

We build two trees ( $\mathcal{I}_q$  and  $\mathcal{I}_r$ ).

We'll traverse the trees simultaneously. (This may be dual breadth-first, dual depth-first, or some combination thereof.)

When we visit a node pair (query and reference nodes), we'll see if we can prune that pair. Otherwise, we'll perform some base case between points contained in the two nodes.

**These algorithms are fast.**

# These algorithms are fast

Speedups are over naive algorithm.

- All-nearest-neighbors search: 100-10000x+
- 2-point correlation: 100x-1000x+
- Exact  $k$ -means clustering: 100x+
- Kernel density estimation: 10000x+
- Mean shift: 10x-100x+
- Euclidean minimum spanning tree calculation: 1000x+

# Range search

Given a query set  $S_q$  and a reference set  $S_r$  and a range  $[l, u]$ ,

# Range search

Given a query set  $S_q$  and a reference set  $S_r$  and a range  $[l, u]$ , find the set of all reference points in  $S_r$  in the range  $[l, u]$  from each query point  $p_q \in S_q$ :

$$S[p_q] = \{p_r : p_r \in S_r, l \leq d(p_q, p_r) \leq u\}.$$

Trees provide a useful strategy for fast range search, by pruning away large amounts of work.

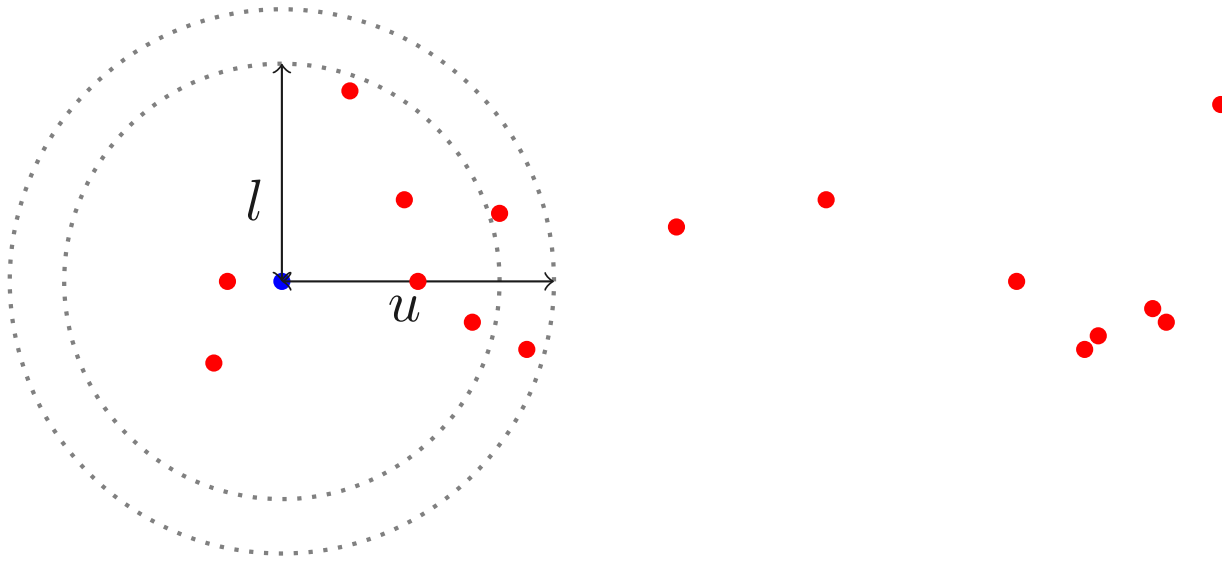


# Range search

Given a query set  $S_q$  and a reference set  $S_r$  and a range  $[l, u]$ , find the set of all reference points in  $S_r$  in the range  $[l, u]$  from each query point  $p_q \in S_q$ :

$$S[p_q] = \{p_r : p_r \in S_r, l \leq d(p_q, p_r) \leq u\}.$$

Trees provide a useful strategy for fast range search, by pruning away large amounts of work.



# Dual $kd$ -tree range search

We need to know when to prune. How?

# Dual $kd$ -tree range search

We need to know when to prune. How?

Use  $d_{\min}(\mathcal{N}_q, \mathcal{N}_r)$  and  $d_{\max}(\mathcal{N}_q, \mathcal{N}_r)$ .

# Dual $kd$ -tree range search

We need to know when to prune. How?

Use  $d_{\min}(\mathcal{N}_q, \mathcal{N}_r)$  and  $d_{\max}(\mathcal{N}_q, \mathcal{N}_r)$ .

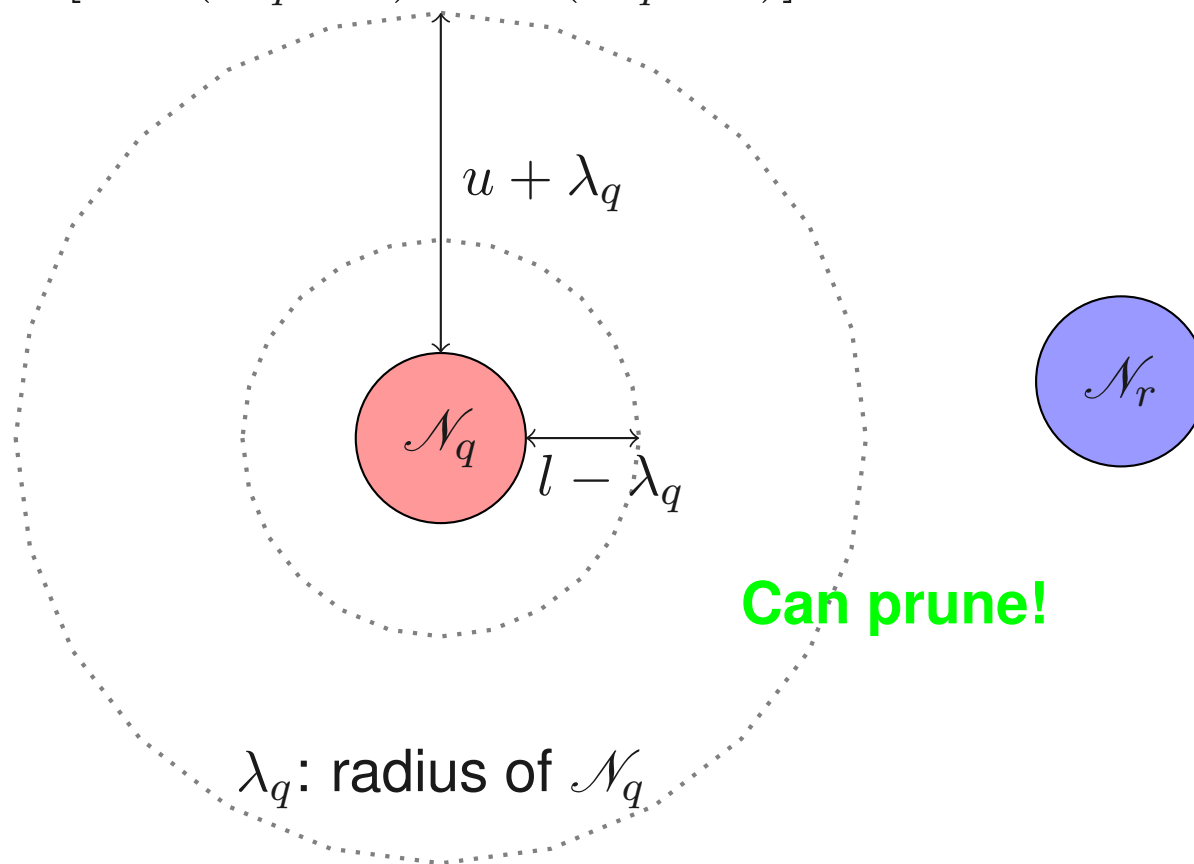
**If  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)]$  does not overlap  $[l, u]$ , we can prune.**

# Dual $kd$ -tree range search

We need to know when to prune. How?

Use  $d_{\min}(\mathcal{N}_q, \mathcal{N}_r)$  and  $d_{\max}(\mathcal{N}_q, \mathcal{N}_r)$ .

**If  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)]$  does not overlap  $[l, u]$ , we can prune.**

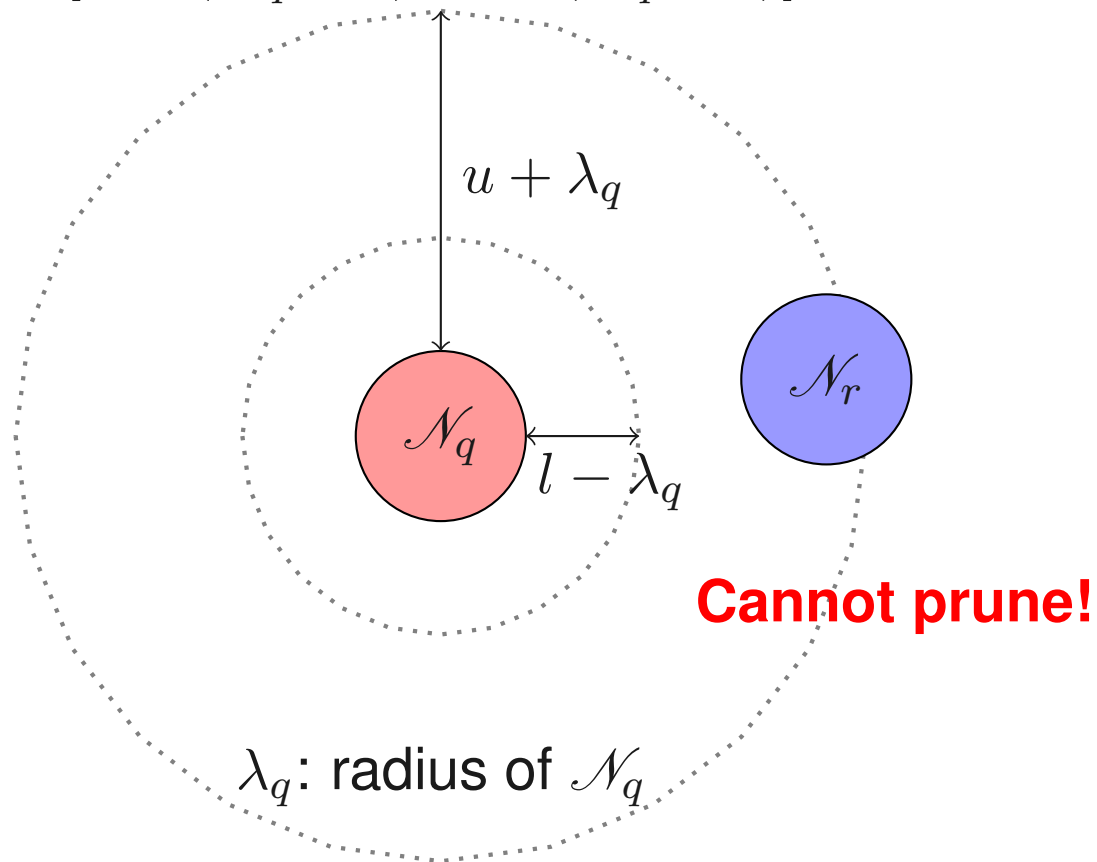


# Dual $kd$ -tree range search

We need to know when to prune. How?

Use  $d_{\min}(\mathcal{N}_q, \mathcal{N}_r)$  and  $d_{\max}(\mathcal{N}_q, \mathcal{N}_r)$ .

**If  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)]$  does not overlap  $[l, u]$ , we can prune.**



# Dual $kd$ -tree range search

**RangeSearchRecursion**( $\mathcal{N}_q$ ,  $\mathcal{N}_r$ )

# Dual $kd$ -tree range search

**RangeSearchRecursion**( $\mathcal{N}_q, \mathcal{N}_r$ )

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!



# Dual $kd$ -tree range search

**RangeSearchRecursion**( $\mathcal{N}_q, \mathcal{N}_r$ )

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!

if  $\mathcal{N}_q$  and  $\mathcal{N}_r$  are leaves:

base case for each  $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$

# Dual $kd$ -tree range search

**RangeSearchRecursion**( $\mathcal{N}_q, \mathcal{N}_r$ )

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!

if  $\mathcal{N}_q$  and  $\mathcal{N}_r$  are leaves:

base case for each  $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$

Base case: calculate  $d(p_q, p_r)$  and see if it lies in the range  $[l, u]$ . If so, add  $p_r$  to  $S[p_q]$ .

# Dual $kd$ -tree range search

**RangeSearchRecursion**( $\mathcal{N}_q, \mathcal{N}_r$ )

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!

if  $\mathcal{N}_q$  and  $\mathcal{N}_r$  are leaves:

base case for each  $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$

# Dual $kd$ -tree range search

**RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r$ )**

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!

if  $\mathcal{N}_q$  and  $\mathcal{N}_r$  are leaves:

base case for each  $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$

else if  $\mathcal{N}_q$  is a leaf:

RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r.\text{left}$ )

RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r.\text{right}$ )

# Dual $kd$ -tree range search

**RangeSearchRecursion**( $\mathcal{N}_q, \mathcal{N}_r$ )

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!

if  $\mathcal{N}_q$  and  $\mathcal{N}_r$  are leaves:

base case for each  $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$

else if  $\mathcal{N}_q$  is a leaf:

RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r.\text{left}$ )

RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r.\text{right}$ )

else if  $\mathcal{N}_r$  is a leaf:

RangeSearchRecursion( $\mathcal{N}_q.\text{left}, \mathcal{N}_r$ )

RangeSearchRecursion( $\mathcal{N}_q.\text{right}, \mathcal{N}_r$ )

# Dual $kd$ -tree range search

**RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r$ )**

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)] \cap [l, u] = \emptyset$ :

return; // Pruned!

if  $\mathcal{N}_q$  and  $\mathcal{N}_r$  are leaves:

base case for each  $p_q \in \mathcal{N}_q, p_r \in \mathcal{N}_r$

else if  $\mathcal{N}_q$  is a leaf:

RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r.\text{left}$ )

RangeSearchRecursion( $\mathcal{N}_q, \mathcal{N}_r.\text{right}$ )

else if  $\mathcal{N}_r$  is a leaf:

RangeSearchRecursion( $\mathcal{N}_q.\text{left}, \mathcal{N}_r$ )

RangeSearchRecursion( $\mathcal{N}_q.\text{right}, \mathcal{N}_r$ )

else:

RangeSearchRecursion( $\mathcal{N}_q.\text{left}, \mathcal{N}_r.\text{left}$ )

RangeSearchRecursion( $\mathcal{N}_q.\text{left}, \mathcal{N}_r.\text{right}$ )

RangeSearchRecursion( $\mathcal{N}_q.\text{right}, \mathcal{N}_r.\text{left}$ )

RangeSearchRecursion( $\mathcal{N}_q.\text{right}, \mathcal{N}_r.\text{right}$ )

Now leaving  $\mathcal{U}_{\text{temp}}$  for good...

# n-point correlation function estimation

## 2-point correlation using *kd*-trees.

```
TwoPoint(QNODE,DNODE,r)
  if excludes(QNODE,DNODE,r), return;

  if subsumes(QNODE,DNODE,r)
    total = total + ( count(QNODE) × count(DNODE) ); return;

  if leaf(QNODE) and leaf(DNODE)
    if distance(QNODE,DNODE) < r, total = total + 1;

  if leaf(QNODE) and notleaf(DNODE)
    TwoPoint(QNODE,leftchild(DNODE),r); TwoPoint(QNODE,rightchild(DNODE),r);

  if notleaf(QNODE) and leaf(DNODE)
    TwoPoint(leftchild(QNODE),DNODE,r); TwoPoint(rightchild(QNODE),DNODE,r);

  if notleaf(QNODE) and notleaf(DNODE)
    TwoPoint(leftchild(QNODE),leftchild(DNODE),r); TwoPoint(leftchild(QNODE),rightchild(DNODE),r);
    TwoPoint(rightchild(QNODE),leftchild(DNODE),r); TwoPoint(rightchild(QNODE),rightchild(DNODE),r);
```



# Cover tree allnn

Nearest neighbor search using cover trees.

---

**Algorithm 5 Find-All-Nearest** (query cover tree  $p_j$ , cover set  $Q_i$ )

---

- (1) if  $i = -\infty$  then for each  $a \in L(p_j)$   
return  $\arg \min_{b \in Q_{-\infty}} d(a, b)$  as the nearest neighbor of  $a$ .
  - (2) else
    - (a) if  $j < i$  then
      - (i) Set  $Q = \{ \text{Children}(q) : q \in Q_i \}$ .
      - (ii) Set  $Q_{i-1} = \{ q \in Q : d(p_j, q) \leq \min_{q \in Q} d(p_j, q) + 2^i + 2^{j+2} \}$ .
      - (iii) **Find-All-Nearest** ( $p_j, Q_{i-1}$ )
    - (b) else for each  $q_{j-1} \in \text{Children}(p_j)$   
**Find-All-Nearest** ( $q_{j-1}, Q_i$ )
-

# Kernel density estimation

## Kernel density estimation using $kd$ -trees.

```

Dualtree( $\mathcal{P}$ )
while !empty( $\mathcal{P}$ ),
  if  $\frac{|\hat{L}_Q^u - \hat{L}_Q^l|}{|\hat{L}_Q^l|} < \epsilon$ , return.
   $\{Q, T, dl, du, p\} = \text{minpriority}(\mathcal{P})$ .
  if  $p < p_{\max}$ ,
     $dl' = N_T K(\text{maxdist}(Q, T))$ .
     $du' = N_T K(\text{mindist}(Q, T))$ .
     $dl = dl', du = du' - N_T$ .
    if  $\frac{|du' - dl'|}{|l_Q + dl'|} < \delta$ ,
      foreach  $\underline{x}_q \in Q$ ,  $l_q += dl, u_q += du$ .
       $\hat{L}_Q^l -= N_Q \log(l_Q), \hat{L}_Q^u -= N_Q \log(u_Q)$ .
       $l_Q += dl, u_Q += du$ .
       $\hat{L}_Q^l += N_Q \log l_Q, \hat{L}_Q^u += N_Q \log u_Q$ .
      enqueue( $\{Q, T, dl, du, \text{priority}(Q, T) + P\}$ ).
      continue.
    else,
       $dl = -dl, du = -du$ .
      foreach  $\underline{x}_q \in Q$ ,  $l_q += dl, u_q += du$ .
       $\hat{L}_Q^l -= N_Q \log(l_Q), \hat{L}_Q^u -= N_Q \log(u_Q)$ .
       $l_Q += dl, u_Q += du$ .
       $\hat{L}_Q^l += N_Q \log l_Q, \hat{L}_Q^u += N_Q \log u_Q$ .
      if leaf( $Q$ ) and leaf( $T$ ), Dualtree.base( $Q, T$ ).
      enqueue( $Q.\text{left}, T.\text{left}, dl, du, \text{priority}(Q, T)$ ).
      enqueue( $Q.\text{left}, T.\text{right}, dl, du, \text{priority}(Q, T)$ ).
      enqueue( $Q.\text{right}, T.\text{left}, dl, du, \text{priority}(Q, T)$ ).
      enqueue( $Q.\text{right}, T.\text{right}, dl, du, \text{priority}(Q, T)$ ).

Dualtree.base( $Q, T$ )
foreach  $\underline{x}_q \in Q$ ,
  foreach  $\underline{x}_t \in T$ ,
     $c = K(\|\underline{x}_q - \underline{x}_t\|), l_q += c, u_q += c$ .
   $u_q -= N_T$ .
   $\hat{L}_Q^l -= N_Q \log l_Q, \hat{L}_Q^u -= N_Q \log u_Q$ .
   $l_Q = \min_{q \in Q} l_q, u_Q = \max_{q \in Q} u_q - N_T$ .
   $\hat{L}_Q^l += N_Q \log l_Q, \hat{L}_Q^u += N_Q \log u_Q$ .
  
```

# Euclidean MST calculation

Euclidean minimum spanning tree calculation using cover trees.

---

**Algorithm 3** FindComponentNeighbors(Cover tree node  $q_j$ , Reference Set  $R_i$ , Edge set  $e$ )

---

```
    if  $i = -\infty$  then
        // base case
3:   for all  $q$  that are descendants of  $q_j$  and  $r \in R_i$  with
         $r \neq q$  do
            if  $d(q, r) < d(C_q)$  then
                 $d(C_q) = d(q, r)$ ,  $e(C_q) = (q, r)$ 
6:   end if
        end for
    else if  $j < i$  then
9:   // reference descend
         $R = \{r \in \text{Children}(r') : r' \in R_i \text{ and } r \not\sim q_j\}$ 
        
$$d = \min \left\{ d(C_q), \min_{\substack{r \in R \\ r \sim q_j}} \{d(q_j, r) + 2^i\}, \min_{\substack{r \in R \\ r \not\sim q_j}} \{d(q_j, r)\} \right\}$$

12:   $R_{i-1} = \{r \in R : d(q_j, r) \leq d + 2^i + 2^{j+2}\}$ 
         $d(C_q) = d$ 
        FindComponentNeighbors( $q_j, R_{i-1}, e$ )
15: else
        // query descend
        for all  $p_{j-1} \in \text{Children}(q_j)$  do
18:   FindComponentNeighbors( $p_{j-1}, R_i, e$ )
        end for
    end if
```

---

# Can we generalize?

- D. Lee, A.G. Gray, and A.W. Moore. “Dual-tree fast Gauss transforms,” in *Advances in Neural Information Processing Systems 18 (NIPS '05)*, pp. 747–754, 2006.
- M. Klaas, M. Briers, N. de Freitas, A. Doucet, S. Maskell, and D. Lang. “Fast particle smoothing: if I had a million particles,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 481–488, 2006.
- A. Beygelzimer, S.M. Kakade, and J. Langford. “Cover trees for nearest neighbor,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–106, 2006.
- M.P. Holmes, A.G. Gray, C.L. Isbell, Jr. “Fast nonparametric conditional density estimation,” in *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI '07)*, 2007.
- W.B. March, P. Ram, and A.G. Gray. “Fast Euclidean minimum spanning tree: algorithm, analysis, and applications,” in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pp. 603–612, 2010.
- W.B. March, A.J. Connolly, and A.G. Gray. “Fast algorithms for comprehensive n-point correlation estimates,” in *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '12)*, pp. 1478–1486, 2012.
- L. Van Der Maaten. “Accelerating t-SNE using tree-based algorithms,” in *The Journal of Machine Learning Research* 15.1, pp. 3221–3245, 2014.
- R.R. Curtin, P. Ram. “Dual-tree fast max-kernel search,” in *Statistical Analysis and Data Mining*, volume 7, issue 4, pp. 229–253, 2014.
- M. Vladymyrov and M.A. Carriera-Perpinán. “Linear-time training of nonlinear low-dimensional embeddings,” in *Proceedings of The 33rd International Conference on Artificial Intelligence and Statistics (AISTATS 2014)*, 2014.
- R.R. Curtin. “Faster dual-tree traversal for nearest neighbor search”, in *Similarity Search and Applications (SISAP 2015)*, 2015, p. 77-89.
- R.R. Curtin. “A dual-tree algorithm for fast k-means clustering with large k”, In *Proceedings of the 2017 SIAM International Conference on Data Mining (SDM '17)*, p. 300-308, 2017. 2017.
- M. Kumar, R.R. Curtin. “Accelerating LMNN with trees”. *In preparation...*

# Can we generalize?

D. Lee, A.G. Gray, and A.W. Moore. “Dual-tree fast Gauss transforms,” in *Advances in Neural Information Processing Systems 18 (NIPS '05)*, pp. 747–754, 2006.

M. Klaas, M. Briers, N. de Freitas, A. Doucet, S. Maskell, and D. Lang. “Fast particle smoothing: if I had a million particles,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 481–488, 2006.

A. Beygelzimer, S.M. Kakade, and J. Langford. “Cover trees for nearest neighbor,” in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–106, 2006.

M.P. Holmes, A.G. Gray, C.L. Isbell, Jr. “Fast nonparametric conditional density estimation,” in *Proceedings of the 23rd Conference on Uncertainty in Artificial Intelligence (UAI '07)*, 2007.

W.B. March, P. Ram, and A.G. Gray. “Fast Euclidean minimum spanning tree: algorithm, analysis, and applications,” in *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*, pp. 200–210, 2010.

W.B. March, A.J....

**Yes! We can generalize!**

...dings of the 18th

L. Van Der Maaten. “Accelerating t-SNE using tree-based algorithms,” in *The Journal of Machine Learning Research* 15.1, pp. 3221–3245, 2014.

R.R. Curtin, P. Ram. “Dual-tree fast max-kernel search,” in *Statistical Analysis and Data Mining*, volume 7, issue 4, pp. 229–253, 2014.

M. Vladymyrov and M.A. Carriera-Perpinán. “Linear-time training of nonlinear low-dimensional embeddings,” in *Proceedings of The 33rd International Conference on Artificial Intelligence and Statistics (AISTATS 2014)*, 2014.

R.R. Curtin. “Faster dual-tree traversal for nearest neighbor search”, in *Similarity Search and Applications (SISAP 2015)*, 2015, p. 77-89.

R.R. Curtin. “A dual-tree algorithm for fast k-means clustering with large k”, in *Proceedings of the 2017 SIAM International Conference on Data Mining (SDM '17)*, p. 300-308, 2017. 2017.

M. Kumar, R.R. Curtin. “Accelerating LMNN with trees”. *In preparation...*

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :
  - performs a computation to assign a score to the combination



# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :
  - performs a computation to assign a score to the combination—`Score()`

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :
  - performs a computation to assign a score to the combination— $\text{Score}()$
  - if the score is  $\infty$ , the combination is pruned

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :
  - performs a computation to assign a score to the combination— $\text{Score}()$
  - if the score is  $\infty$ , the combination is pruned
  - otherwise a computation is performed between each point in  $\mathcal{N}_q$  and  $\mathcal{N}_r$

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :
  - performs a computation to assign a score to the combination—`Score()`
  - if the score is  $\infty$ , the combination is pruned
  - otherwise a computation is performed between each point in  $\mathcal{N}_q$  and  $\mathcal{N}_r$ —`BaseCase()`

# Dual-tree traversals

A **pruning dual-tree traversal** is a process that, given two trees  $\mathcal{T}_q$  and  $\mathcal{T}_r$ :

- will visit combinations of nodes  $(\mathcal{N}_q, \mathcal{N}_r)$
- when visiting a combination  $(\mathcal{N}_q, \mathcal{N}_r)$ :
  - performs a computation to assign a score to the combination— $\text{Score}()$
  - if the score is  $\infty$ , the combination is pruned
  - otherwise a computation is performed between each point in  $\mathcal{N}_q$  and  $\mathcal{N}_r$ — $\text{BaseCase}()$
- if no nodes are pruned,  $\text{BaseCase}()$  is called with each point in the query tree and each point in the reference tree

R.R. Curtin, W.B. March, P. Ram, D.V. Anderson, A.G. Gray, C.L. Isbell, Jr. "Tree-independent dual-tree algorithms," in *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, pp. 1435–1443, Atlanta, GA, 2013.

# Putting it all together

A type of space tree

# Putting it all together

A type of space tree +

A pruning dual-tree traversal

# Putting it all together

A type of space tree +

A pruning dual-tree traversal +

Problem-specific BaseCase() and Score() functions



# Putting it all together

A type of space tree +

A pruning dual-tree traversal +

Problem-specific BaseCase() and Score() functions =

## **Dual-tree algorithm**

What does this get us?

# Putting it all together

A type of space tree +  
A pruning dual-tree traversal +  
Problem-specific BaseCase() and Score() functions =

## Dual-tree algorithm

What does this get us?

- Easy-to-understand, beautiful algorithms.
- Improvements to one component propagate!
- Simple development of new algorithms.
- Improved software implementations and abstractions.

R.R. Curtin, W.B. March, P. Ram, D.V. Anderson, A.G. Gray, C.L. Isbell, Jr. “Tree-independent dual-tree algorithms,” in *Proceedings of the 30th International Conference on Machine Learning (ICML '13)*, pp. 1435–1443, Atlanta, GA, 2013.

# Tree-independent range search

**BaseCase**( $p_q, p_r$ ):

$d \leftarrow d(p_q, p_r)$

if  $d \in [l, u]$ :

    add  $p_r$  to list of results for  $p_q$

# Tree-independent range search

**BaseCase**( $p_q, p_r$ ):

$d \leftarrow d(p_q, p_r)$

if  $d \in [l, u]$ :

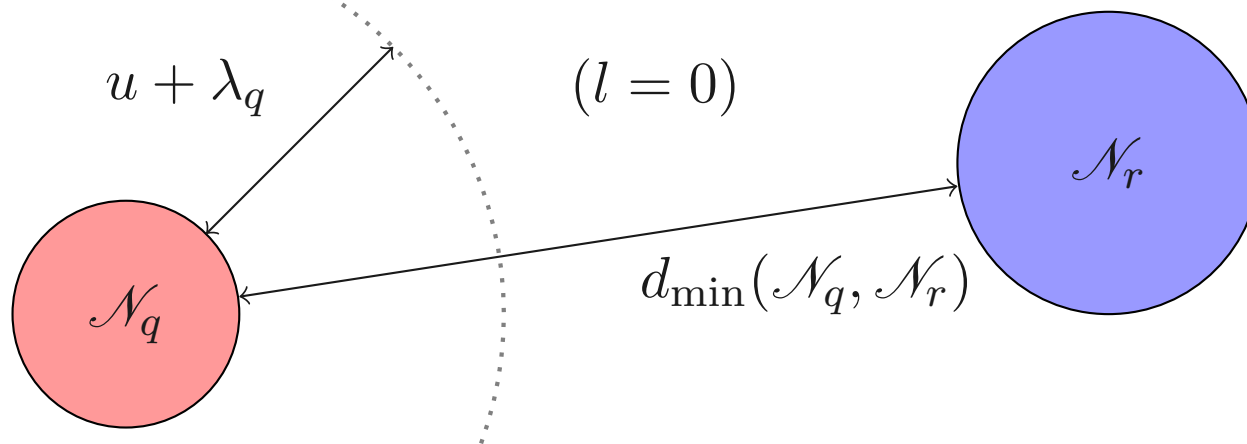
add  $p_r$  to list of results for  $p_q$

**Score**( $\mathcal{N}_q, \mathcal{N}_r$ ):

if  $[d_{\min}(\mathcal{N}_q, \mathcal{N}_r), d_{\max}(\mathcal{N}_q, \mathcal{N}_r)]$  does not  
overlap  $[l, u]$ :

return  $\infty$  // Pruned!

return 0 // Recursion order doesn't matter.



# Another quick stopping point for questions

(don't worry, we are now at the results part)

# Theory

**Theory for trees is hard!** We want to (ideally) say something like “nearest neighbor search takes  $O(\log N)$  time with trees”. But we can’t:

- I can build an adversarial dataset where you can never prune a node.
- How deep is your tree?
- What is the ratio of a child node’s volume to a parent node’s volume?
- How many descendant points are held in a child vs. in its parent?

With a *kd*-tree, we have a lot of problems!

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$



# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$

- Bounded width: each node has up to  $c^4$  children.

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$

- Bounded width: each node has up to  $c^4$  children.
- Bounded depth: each node has depth  $O(c^2 \log N)$ .

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$

- Bounded width: each node has up to  $c^4$  children.
- Bounded depth: each node has depth  $O(c^2 \log N)$ .
- Bounded tree size: a cover tree has  $O(N)$  nodes.

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$

- Bounded width: each node has up to  $c^4$  children.
- Bounded depth: each node has depth  $O(c^2 \log N)$ .
- Bounded tree size: a cover tree has  $O(N)$  nodes.
- Bounded construction time:  $O(c^6 N \log N)$ .

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$

- Bounded width: each node has up to  $c^4$  children.
- Bounded depth: each node has depth  $O(c^2 \log N)$ .
- Bounded tree size: a cover tree has  $O(N)$  nodes.
- Bounded construction time:  $O(c^6 N \log N)$ .
- Bounded NN search time:  $O(c^{12} \log N)$  for a single query point.

A. Beygelzimer, S.M. Kakade, and J. Langford. "Cover trees for nearest neighbor," in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–106, 2006.

# The cover tree

To answer these questions, we need a weird structure called the **cover tree**. It is very complex and I don't want to talk about how you build one unless forced to...!

Given a dataset  $S$  with  $|S| = N$ , the *expansion constant* is the smallest  $c \geq 2$  such that for all  $p \in S$ ,

$$|B_S(p, 2\delta)| \leq c|B_S(p, \delta)|.$$

- Bounded width: each node has up to  $c^4$  children.
- Bounded depth: each node has depth  $O(c^2 \log N)$ .
- Bounded tree size: a cover tree has  $O(N)$  nodes.
- Bounded construction time:  $O(c^6 N \log N)$ .
- ~~Bounded NN search time:  $O(c^{12} \log N)$  for a single query point.~~

A. Beygelzimer, S.M. Kakade, and J. Langford. "Cover trees for nearest neighbor," in *Proceedings of the 23rd International Conference on Machine Learning (ICML '06)*, pp. 97–106, 2006.

# Generalized runtime bound

**A dual-tree algorithm using cover trees and the cover tree pruning dual-tree traversal takes time bounded by**

$$O(c_q^4 \psi \chi |R^*| (N + i(\mathcal{T}_q) + \theta)).$$

- $N$ : number of query points
- $i(\mathcal{T}_q)$ : imbalance of query tree
- $\theta$ : measure of scale difference between query and reference sets
- $c_r$ : expansion constant of reference set
- $|R^*|$ : size of largest set of reference nodes encountered during the traversal
- $\psi$ : running time of BaseCase()
- $\chi$ : running time of Score()

R.R. Curtin, D. Lee, W.B. March, P. Ram. "Plug-and-play runtime analysis for dual-tree algorithms," *The Journal of Machine Learning Research*, vol. 16, p. 3269-3297, 2015.

# Applications

- Monochromatic nearest neighbor search:

$$O(c_r^9(N + i_t(\mathcal{T})))$$

- Nearest neighbor search:

$$O(c_r^4 c_{qr}^5 (N + i_t(\mathcal{T}_q) + \theta))$$

- Approx. kernel density estimation:

$$O(c_r^{8 + \lceil \log_2 \zeta \rceil} (N + i_t(\mathcal{T}_q) + \theta))$$

- Range search: (under some assumptions)

$$O(c_r^{8 + \beta} (N + i_t(\mathcal{T}_q) + \theta))$$

- Sparse kernel matrix approximation:

$$O(c_r^{7 + \lceil \log_2 \nu \rceil} (N + i_t(\mathcal{T}_q)))$$

- Fast max-kernel search:

$$O(\gamma_r c_r^{7 \log_2 \alpha} (N + i_t(\mathcal{T}_q) + \theta))$$

R.R. Curtin, D. Lee, W.B. March, P. Ram. "Plug-and-play runtime analysis for dual-tree algorithms," *The Journal of Machine Learning Research*, vol. 16, p. 3269-3297, 2015.



# Empirical results

Algorithm	# Data	Quadratic	Single-tree	Dual-tree	ST Speedup	DT Speedup
twopoint	10,000	132	2.2	1.2	60	110
twopoint	50,000	3300 est.	11.8	7.0	280	471
twopoint	150,000	30899 est.	37	20	835	1545
twopoint	300,000	123599 est.	76	40	1626	3090
nearest	10,000	139	2.0	1.4	70	99
nearest	20,000	556 est.	11.6	9.8	48	57
nearest	50,000	3475 est.	30.6	26.4	114	132
outliers	10,000	141	2.3	1.2	61	118
outliers	50,000	3525 est.	12	6.5	294	542
outliers	150,000	33006 est.	36	21	917	1572
outliers	300,000	132026 est.	72	44	1834	3001

Figure 3: Our experiments timed our algorithms on large astronomical datasets of current scientific interest, consisting of x-y positions of sky objects from the Sloan Digital Sky Survey. All times are given in seconds, and runs were performed on a Pentium III-500 MHz Linux workstation. The larger runtimes for the quadratic algorithm were estimated based on those for smaller datasets. The dual *kd*-tree method is about a factor of 2 faster than the single *kd*-tree method, and both are 3 orders of magnitude faster than the quadratic method for a medium-sized dataset of 300,000 points.

A.G. Gray, A.W. Moore. "N-body problems in statistical learning". Advances in Neural Information Systems Processing (NIPS 2001), 2001.

# Empirical results

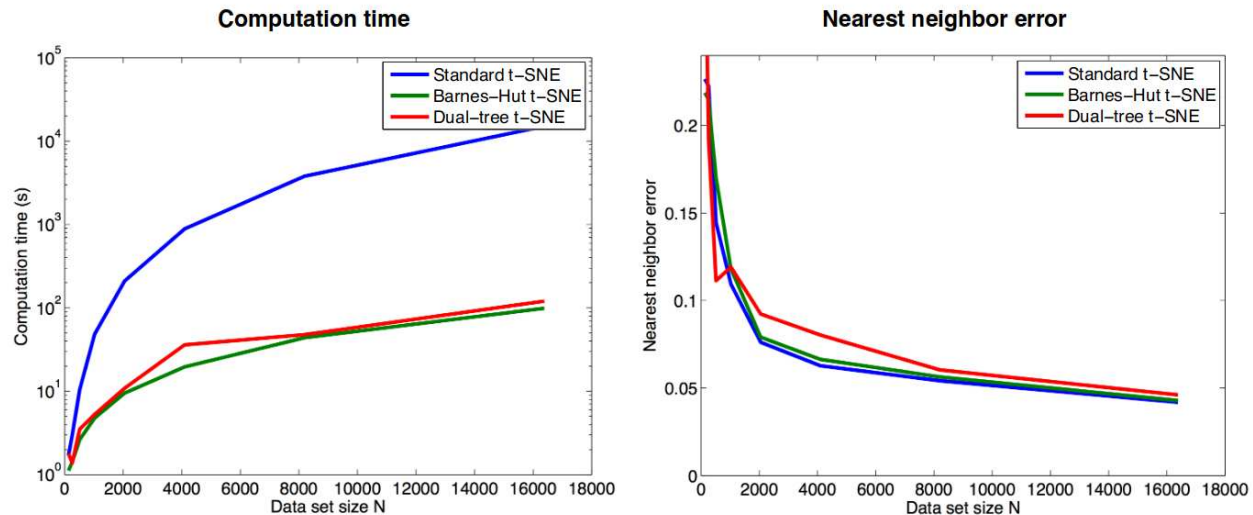
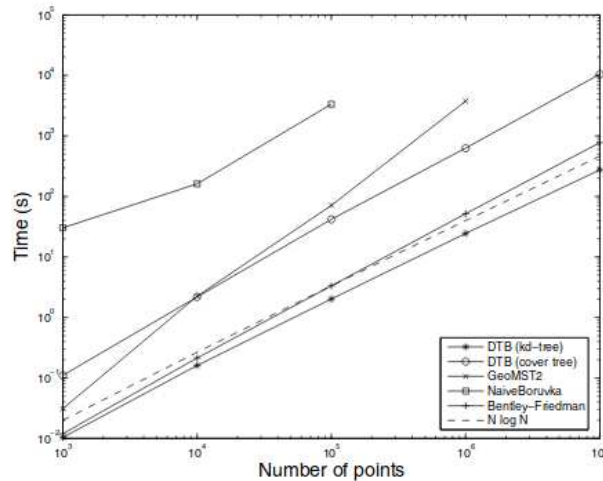


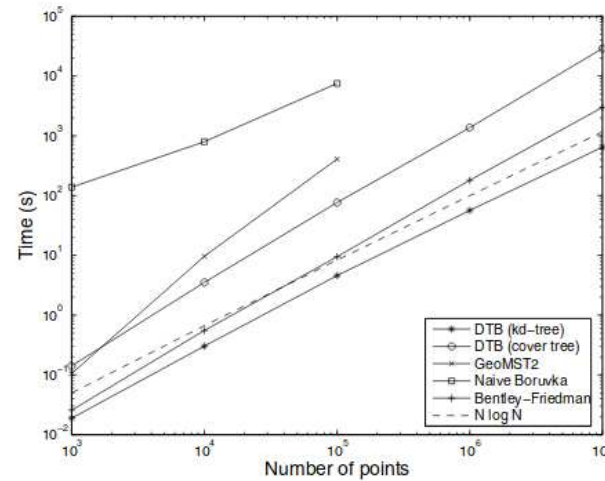
Figure 4: Computation time (in seconds) required to embed MNIST digits (left) and the 1-nearest neighbor errors of the corresponding embeddings (right) as a function of data set size  $N$  for standard t-SNE (in blue), Barnes-Hut t-SNE (in green), and dual-tree t-SNE (in red). Note that the required computation time, which is shown on the  $y$ -axis of the left figure, is plotted on a logarithmic scale.

L. Van Der Maaten. "Accelerating t-SNE using tree-based algorithms." *The Journal of Machine Learning Research* 15.1 (2014): 3221-3245.

# Empirical results



(a) Three-dimensional data generated from a mixture of gaussians.



(b) Four-dimensional SDSS spectra.

**Figure 2: EMST computation times on log-log scale. All timings are in seconds.**

$N$	dim	DTB $kd$	DTB cover	BF [5]
40,000	3840	45825.18	<b>15791.37</b>	45780.43
1,000,000	3	<b>17.39</b>	333.45	42.54

**Table 1: Comparison of DualTreeBoruvka and Bentley-Friedman timings. Timings are in seconds.**

W.B. March, P. Ram, and A.G. Gray. "Fast Euclidean minimum spanning tree: algorithm, analysis, and applications." *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '10)*. ACM, 2010.

# Empirical results

Data = SDSS, $D = 2$				
$N$	$h^*$	Naive Time	Single Tree Time	Dual Tree Time
12.5K	.0025	7	.45	.12
25K	.0025	31	1.4	.31
50K	.001	123	2.1	.46
100K	.00075	494	5	1.0
200K	.0005	1976*	10	2
400K	.0005	7904*	27	5
800K	.00025	31616*	49	10
1600K	.00025	126465*	127	23

Table 1: Scaling with dataset size: numerical values

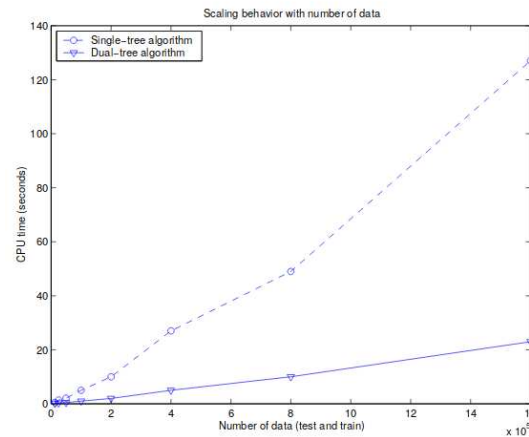


Figure 4: Scaling with dataset size: plot of values

A.G. Gray and A.W. Moore. "Nonparametric density estimation: Toward computational tractability." *Proceedings of the 2003 SIAM International Conference on Data Mining (SDM '03)*. Society for Industrial and Applied Mathematics, 2003.

# Empirical results

algorithm	covertime	power	lcdm	sdss-dr6
<i>kd</i> -tree, unordered	302.8s (1.09B)	1163.0s (18.7B)	5628.7s (41.5B)	24717s (156B)
<i>kd</i> -tree, prioritized	15.823s (52.5M)	30.072s (302M)	319.871s (1.87B)	9069s (50.3B)
<i>kd</i> -tree, improved	<b>4.469s (12.8M)</b>	<b>12.714s (200M)</b>	<b>71.587s (350M)</b>	<b>428.9s (2.14B)</b>
single <i>kd</i> -tree	6.207s (16.3M)	19.684s (232M)	120.6s (476M)	471.4s (2.24B)
ball tree, unordered	163.027s (2.90B)	771.975s (25.3B)	1861.9s (71.1B)	9444s (363B)
ball tree, prioritized	52.487s (902M)	113.437s (3.90B)	386.74s (14.4B)	5202s (192B)
ball tree, improved	<b>27.251s (392M)</b>	<b>83.744s (2.58B)</b>	<b>195.175s (6.46B)</b>	<b>5150s (136B)</b>
single ball tree	29.948s ( <b>228M</b> )	138.422s ( <b>2.49B</b> )	402.6s ( <b>5.93B</b> )	7226s ( <b>101B</b> )

**Table 2.** Runtime (distance evaluations) for exact nearest neighbor search.

R.R. Curtin. "Faster dual-tree traversal for nearest neighbor search." *International Conference on Similarity Search and Applications*. Springer, 2015.

# Conclusions

Let's hope that I convinced you of the following things.

- Geometry is everywhere in machine learning!
- Single-tree and dual-tree algorithms are really not all that complex even though they look that way on paper.
- The class of trees that can be used with a single- or dual-tree algorithm can be concisely expressed.
- In low(ish) dimensionality, trees are super effective at reducing the number of distance computations necessary to solve a problem.\*\*
- Trees don't have great theoretical properties, but sometimes you can say some things about the performance of tree-based algorithms.

And future directions...

# Questions?