The R-Tree

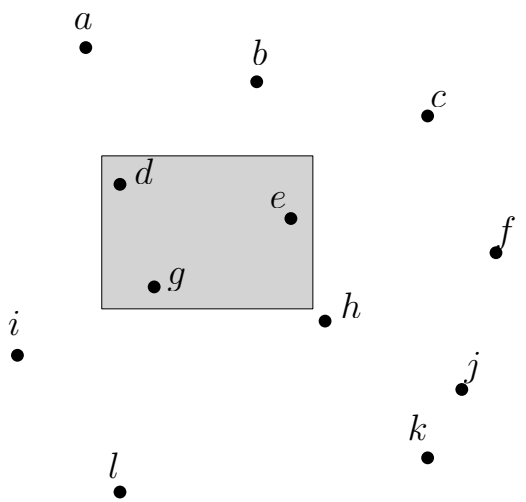
We will study a new structure called the R-tree, which can be thought of as a multi-dimensional extension of the B-tree. The R-tree supports efficiently a variety of queries (as we will find out later in the course), and is implemented in numerous database systems. Our discussion in this lecture will focus on orthogonal range reporting.

2D Orthogonal Range Reporting (Window Query)

Let S be a set of points in $\mathbb{R}^2$. Given an axis-parallel rectangle q, a *range query* returns all the points of S that are covered by q, namely, $S \cap q$.

The definition can be extended to any dimensionality in a straightforward manner.

Example



The result is $\{d, e, g\}$ for the shaded rectangle q.

Applications

- Find all restaurants in the Manhattan area.
- Find all professors whose ages are in $[20, 40]$ and their annual salaries are in $[200k, 300k]$.
- ...

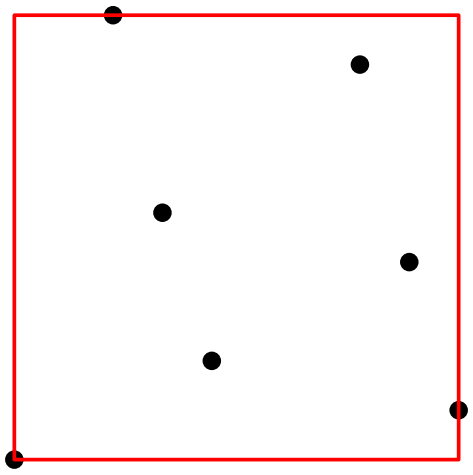
R-Tree

- Each leaf node has between $0.4B$ and B data points, where $B \geq 3$ is a parameter. The only exception applies when the leaf is the root, in which case it is allowed to have between 1 and B points. All the leaf nodes are at the same level.
- Each internal node has between $0.4B$ and B child nodes, except when the node is the root, in which case it needs to have at least 2 child nodes.

In practice, for a disk-resident R-tree, the value of B depends on the block size of the disk so that each node is stored in a block.

R-Tree

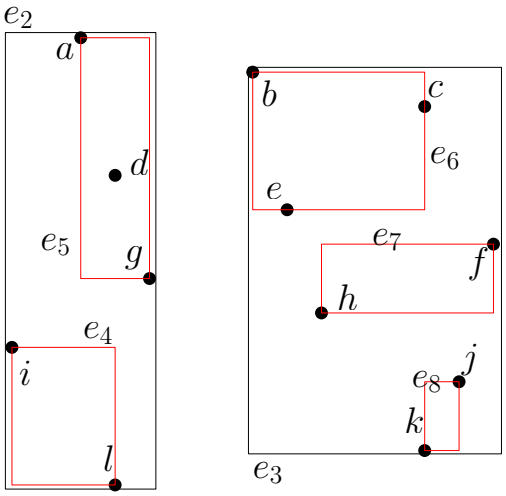
For any node u, denote by S_u the set of points in the subtree of u. Consider now u to be an internal node with child nodes $v_1, ..., v_f$ ($f \leq B$). For each v_i ($i \leq f$), u stores the minimum bounding rectangle (MBR) of S_{v_i}, denoted as $MBR(v_i)$.

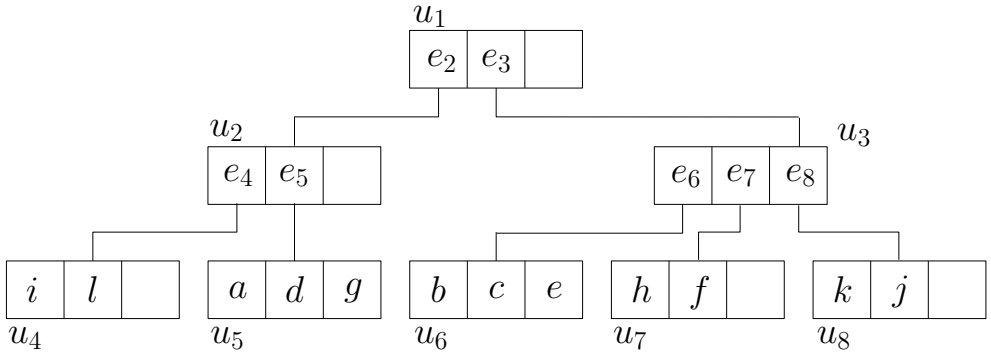


The above is an MBR on 7 points.

Example

Assume $B = 3$.





Answering a Range Query

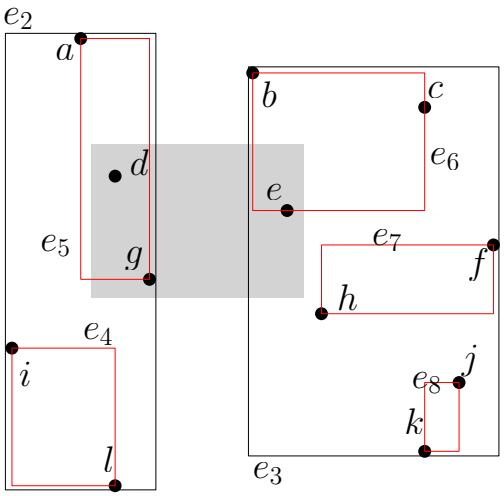
Let q be the search region of a range query. Below we give the pseudo-code of the query algorithm, which is invoked as `range-query`$(root, q)$, where *root* is the root of the tree.

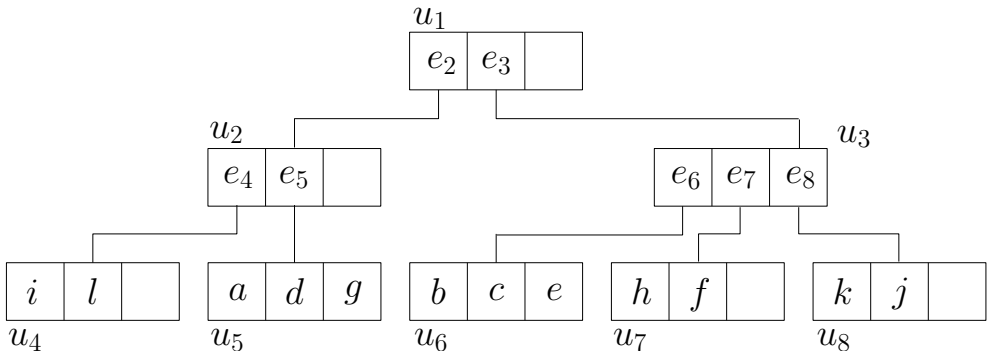
Algorithm `range-query`(u, r)

```
1.  if u is a leaf then
2.      report all points stored at u that are covered by r
3.  else
4.      for each child v of u do
5.          if MBR(v) intersects r then
6.              range-query(v, r)
```

Example

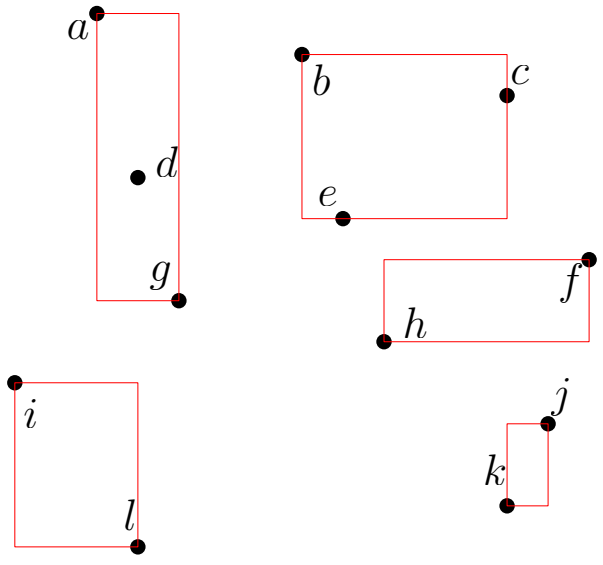
Nodes u_1, u_2, u_3, u_5, u_6 are accessed to answer the query with the shaded search region.





R-Tree Construction Can Be "Arbitrary"

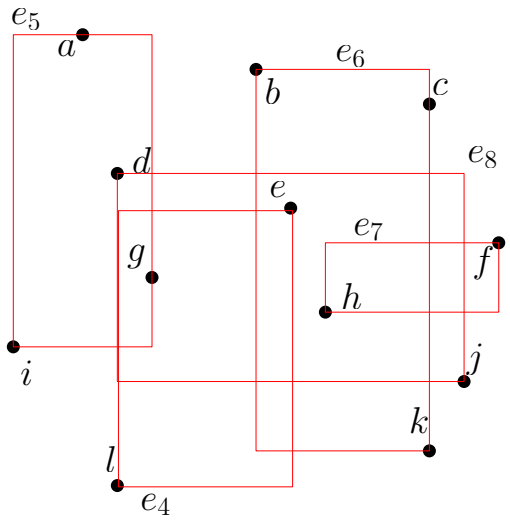
Have you wondered why the leaf nodes are created in this way? For example, is it absolutely necessary to group *i* and *l* into a leaf node?

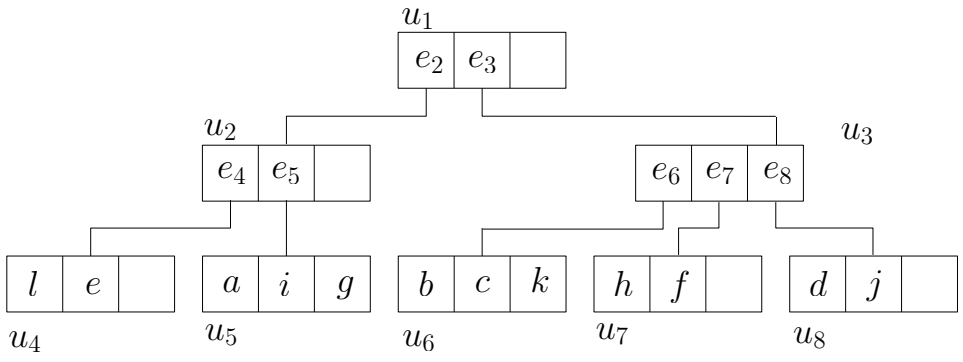


The R-tree definition has no formal constraint whatsoever on the grouping of data into nodes (unlike B-trees), but some R-trees have poorer performance than others; see the next slide.

R-Tree Construction Can Be "Arbitrary"

Is this a good R-tree?



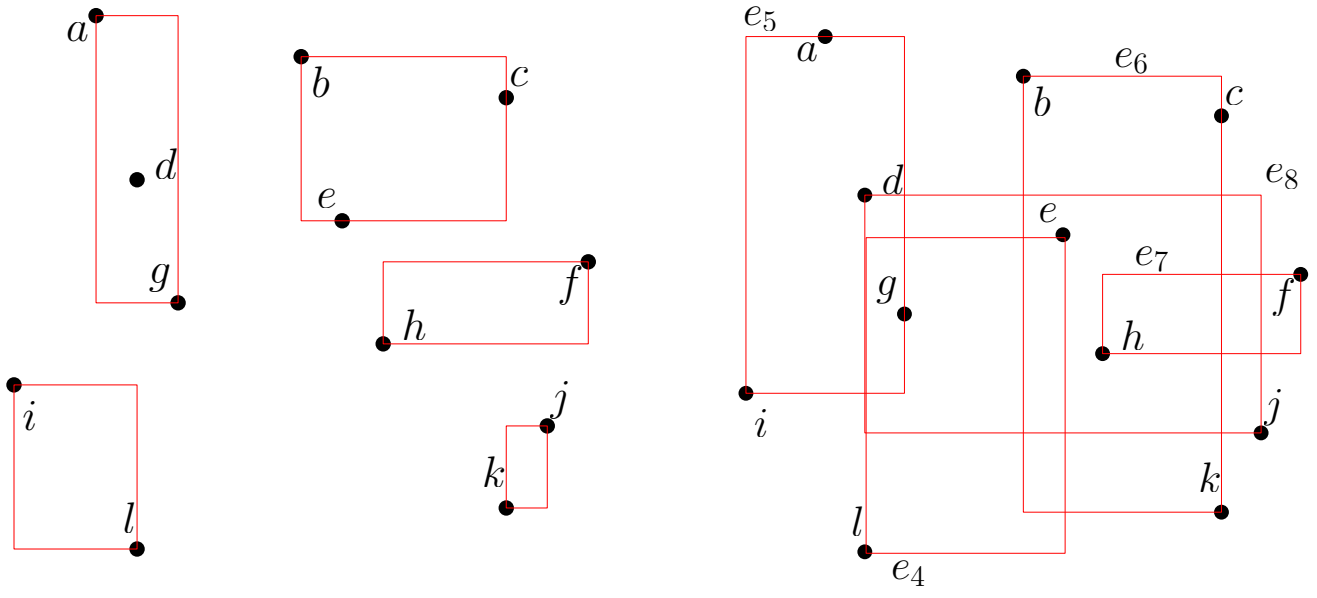


Implication?

R-Tree Construction: A Common Principle

In general, the construction algorithm of the R-tree aims at minimizing the perimeter sum of all the MBRs.

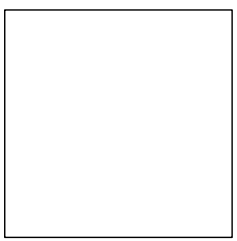
For example, the left tree has a smaller perimeter sum than the right one.



R-Tree Construction: A Common Principle

Why not minimize the area?

A rectangle with a smaller perimeter usually has a smaller area, but not the vice versa. Later in the course, we will see an analysis that formally validates this intuition.



The above two rectangles have the same area.

Insertion

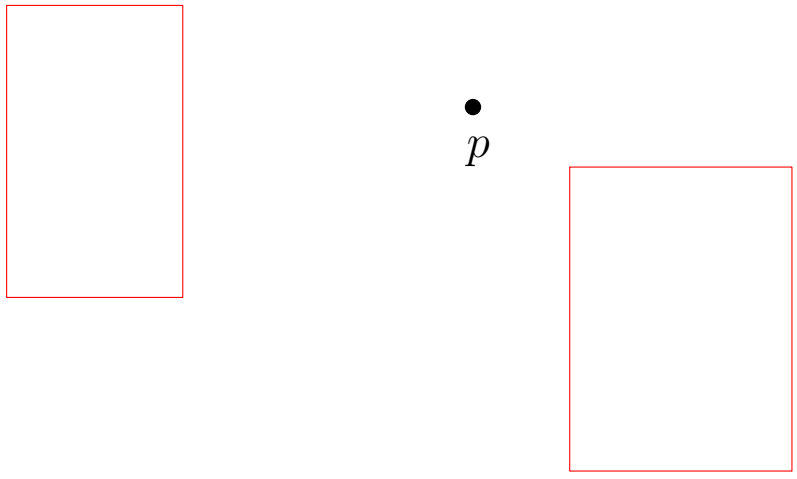
Let p be the point being inserted. The pseudo-code below should is invoked as `insert`$(root, p)$, where *root* is the root of the tree.

Algorithm `insert`(u, p)

1. **if** u is a leaf node **then**
2. add p to u
3. **if** u overflows **then**
   /* namely, u has $B + 1$ points */
4. `handle-overflow`(u)
5. **else**
6. $v \leftarrow$ `choose-subtree`(u, p)
   /* which subtree under u should we insert p into? */
7. `insert`(v, p)

Choose-Subtree

Which MBR would you insert p into?



Algorithm `choose-subtree`(u, p)

1. return the child whose MBR requires the minimum increase in perimeter to cover p.
   break ties by favoring the smallest MBR.

Overflow Handling

Algorithm `handle-overflow`(u)

1. `split`(u) into u and u'
2. **if** u is the root **then**
3. create a new root with u and u' as its child nodes
4. **else**
5. $w \leftarrow$ the parent of u
6. update $MBR(u)$ in w
7. add u' as a child of w
8. **if** w overflows **then**
9. `handle-overflow`(w)

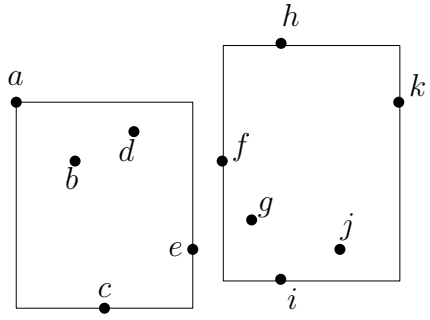
Splitting a Leaf

Essentially we are dealing with the following problem:

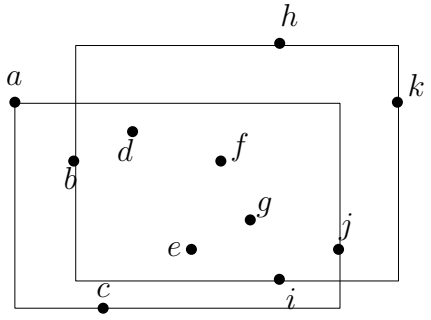
> Let S be a set of $B+1$ points. Divide S into two disjoint sets S_1 and S_2 to minimize the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$, subject to the condition that $|S_1| \geq 0.4B$ and $|S_2| \geq 0.4B$.

Example

The left split is better:



$$S_1 = \{a, b, c, d, e\}$$
$$S_2 = \{f, g, h, i, j, k\}$$



$$S_1 = \{a, d, e, g, j\}$$
$$S_2 = \{b, c, f, h, i, k\}$$

Splitting a Leaf Node

Let $m = |S|$. In 2D space, the leaf-split problem can be solved in $O(m^5)$ time, noticing that each MBR is determined by 4 points.

This, however, is too expensive. In practice, heuristics are used to accelerate the process, but there is no guarantee that we can find the best split — typical "trading quality for efficiency".

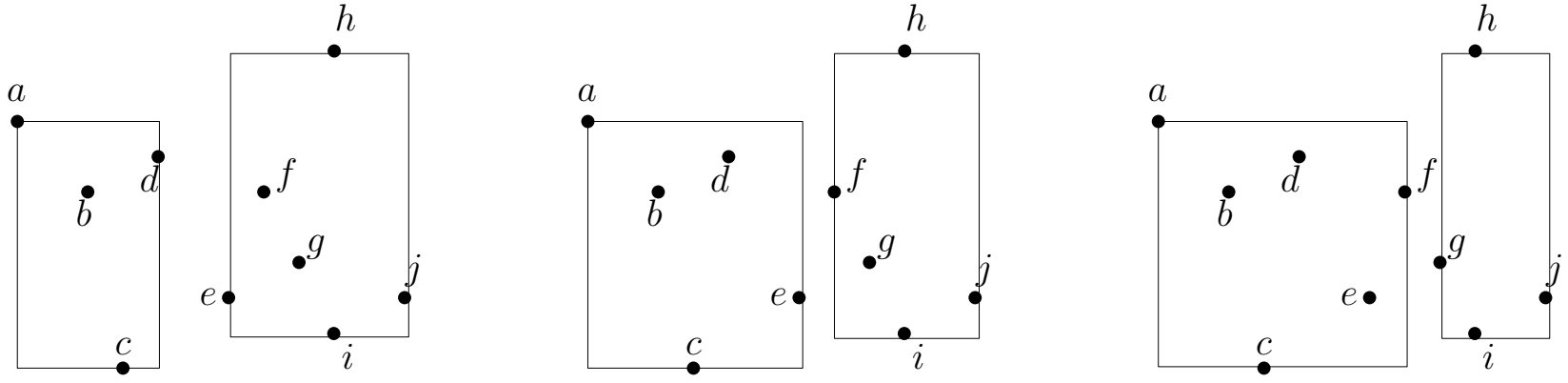
The next slide explains how.

Splitting a Leaf Node

Algorithm `split`(u)

1. m = the number of points in u
2. sort the points of u on x-dimension
3. **for** $i = \lceil 0.4B \rceil$ to $m - \lceil 0.4B \rceil$
4. $\quad S_1 \leftarrow$ the set of the first i points in the list
5. $\quad S_2 \leftarrow$ the set of the other i points in the list
6. $\quad$ calculate the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$; record it if this is the best split so far
7. Repeat Lines 2-6 with respect to y-dimension
8. **return** the best split found

Example



There are 3 possible splits along the x-dimension. Remember that each node must have at least $0.4B = 4$ points (here $B = 10$).

Think:

- How to implement the algorithm in $O(n \log n)$ time?
- Find a counter-example where the algorithm does not give an optimal split.
- We have discussed only the 2D case. How to extend the algorithm to dimensionality $d \geq 3$?

Splitting an Internal Node

Let S be a set of $B+1$ rectangles. Divide S into two disjoint sets S_1 and S_2 to minimize the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$, subject to the condition that $|S_1| \geq 0.4B$ and $|S_2| \geq 0.4B$.

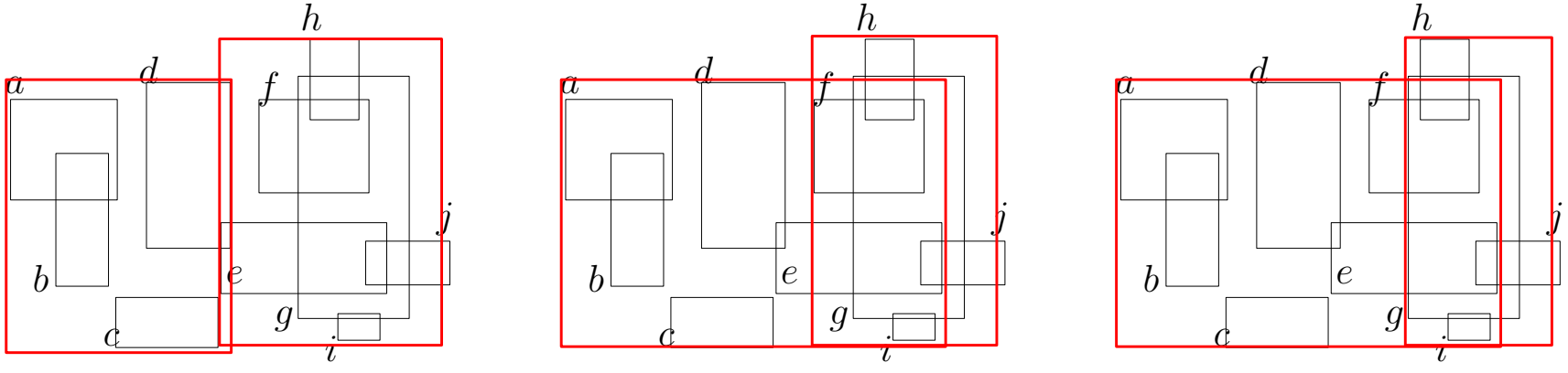
Once again, we will settle for an algorithm that is fast but does not always return an optimal split.

Splitting an Internal Node

Algorithm `split`(u)
/* u is an internal node */

1. m = the number of points in u
2. sort the rectangles in u by their left boundaries on the x-dimension
3. **for** $i = \lceil 0.4B \rceil$ to $m - \lceil 0.4B \rceil$
4. $\quad S_1 \leftarrow$ the set of the first i rectangles in the list
5. $\quad S_2 \leftarrow$ the set of the other i rectangles in the list
6. $\quad$ calculate the perimeter sum of $MBR(S_1)$ and $MBR(S_2)$; record it if this is the best split so far
7. Repeat Lines 2-6 with respect to the right boundaries on the x-dimension
8. Repeat Lines 2-7 w.r.t. the y-dimension
9. **return** the best split found

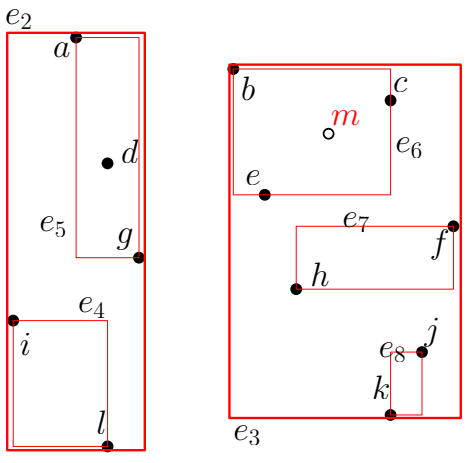
Example

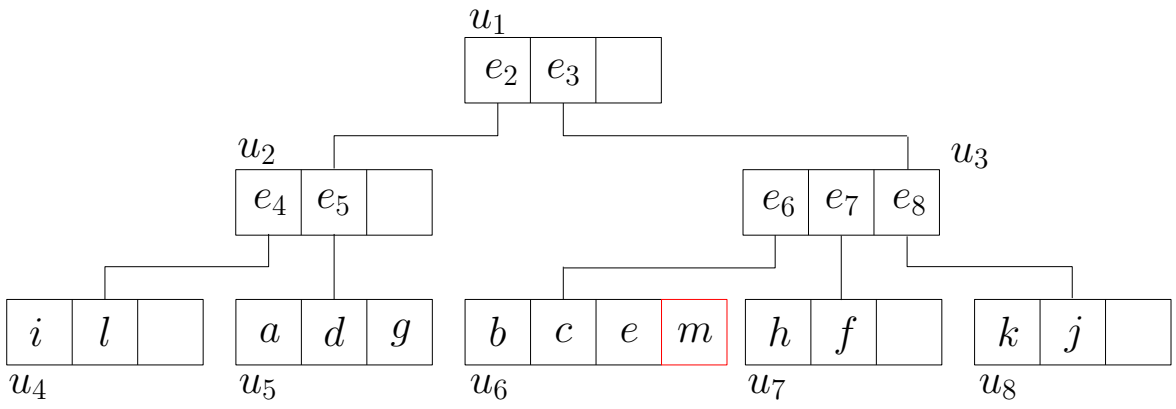


There are 3 possible splits w.r.t. the left boundaries on the x-dimension. Remember that each node must have at least $0.4B = 4$ points (here $B = 10$).

Insertion Example

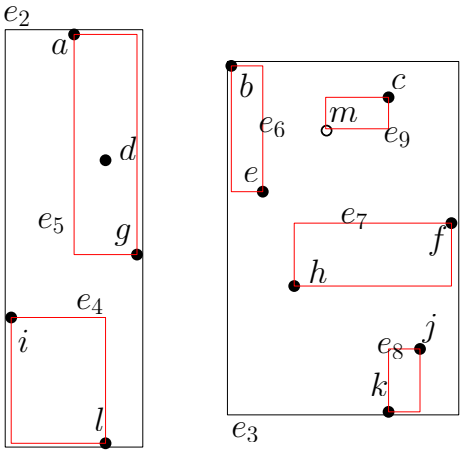
Assume that we want to insert the white point m. By applying `choose-subtree` twice, we reach the leaf node u_6 that should accommodate m. The node overflows after incorporating m (recall $B = 3$).

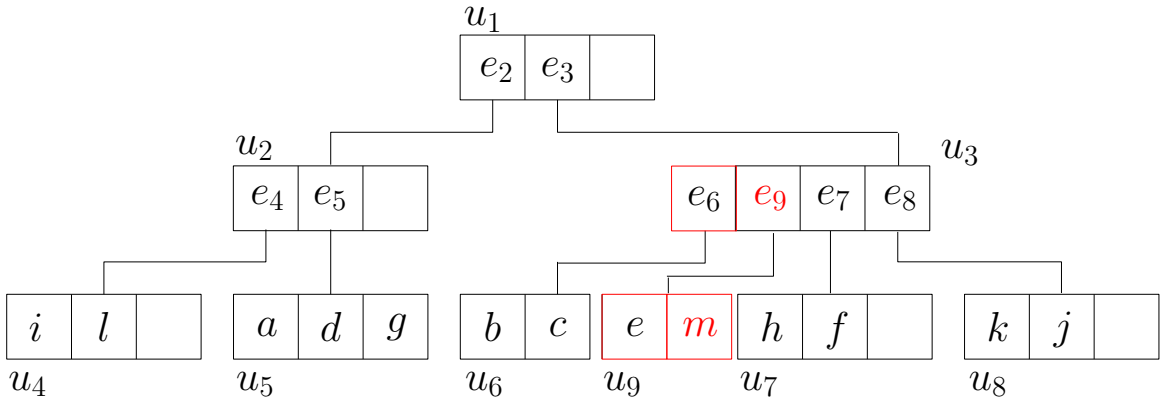




Insertion Example

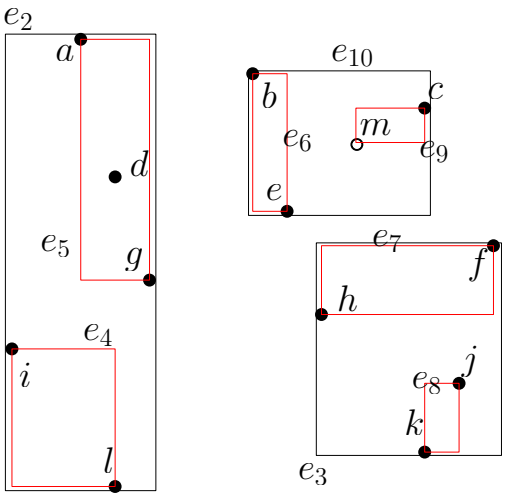
Node u_6 splits, generating u_9. Adding u_9 as a child of u_3 causes u_3 to overflow.

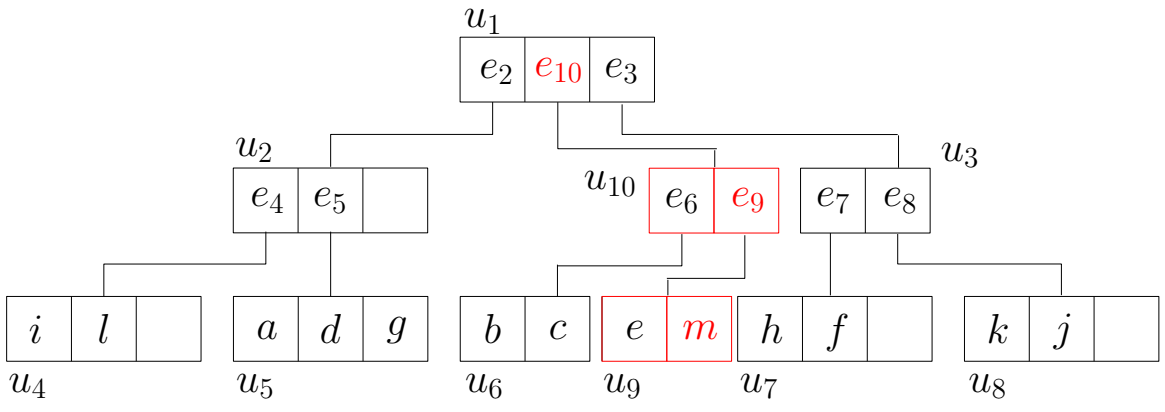




Insertion Example

Node u_3 splits, generating u_{10}. The insertion finishes after adding u_{10} as a child of the root.





Nearest Neighbor Search

In this lecture, we will study a new problem called nearest neighbor search, which plays an important role in a great variety of applications. Our discussion will also introduce two methods: the branch-and-bound and the best first techniques, both of which are generic algorithmic paradigms useful in many scenarios.

Nearest Neighbor Search

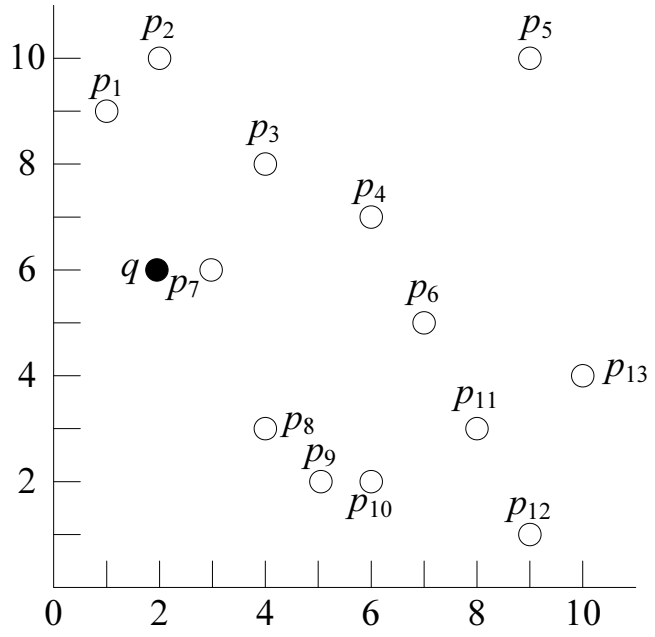
Let P be a set of d-dimensional points in $\mathbb{R}^d$. The (Euclidean) nearest neighbor (NN) of a query point $q \in \mathbb{R}^d$ is the point $p \in P$ that has the smallest Euclidean distance to q.

Given a query point q, an NN query returns the NN(s) of q. Note that multiple points can have the smallest distance to q, in which case they are all nearest neighbors and should be reported.

Note:

- The Euclidean distance between p and q is the length of the line segment connecting p and q.
- We denote the Euclidean distance between p and q as $\|p, q\|$.

Example

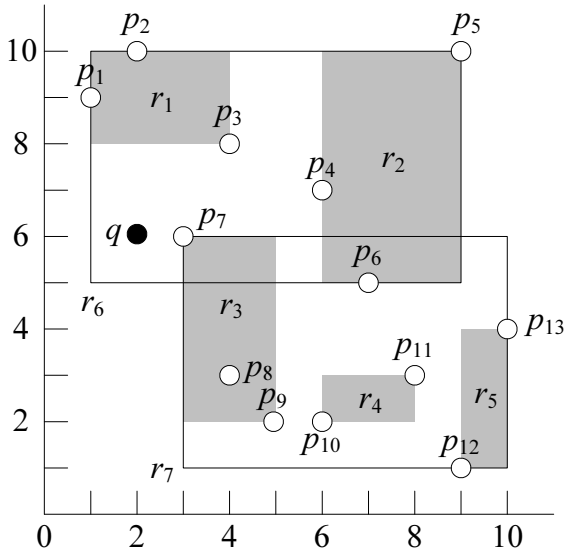


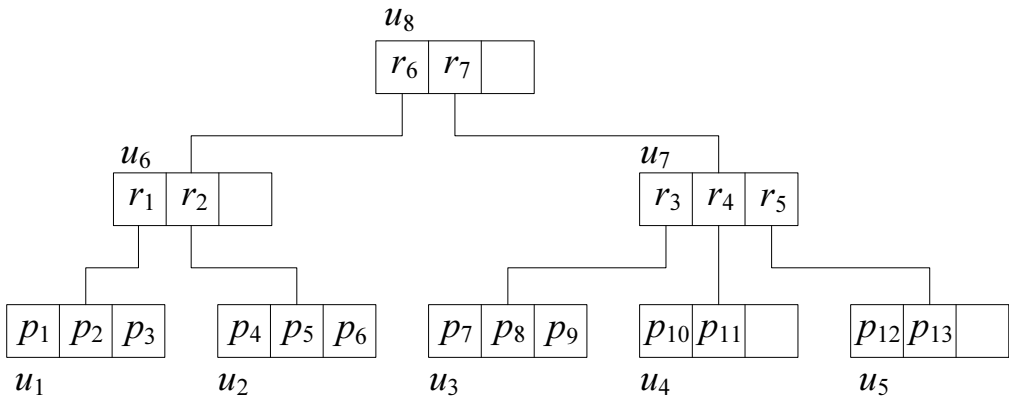
The NN of q is p_7.

Applications

- "Find the McDonald that is nearest to me".
- "Find the customer profile in the database that is most similar to the profile of the new customer".
- "Retrieve the image from the database that is most similar to the one given by the user".
- ...

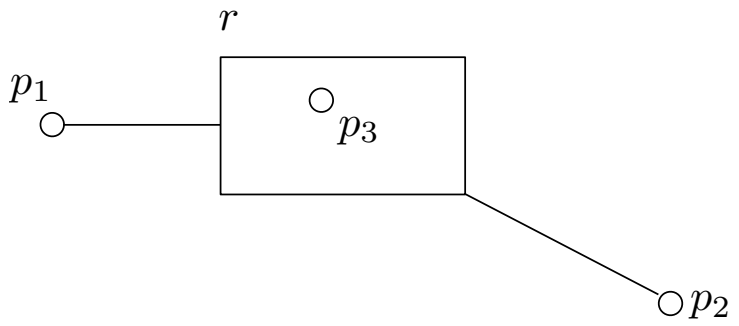
If no pre-processing is allowed on P, we must scan the entire P to answer a NN query. Query efficiency can be significantly improved by using an R-tree on P.





Mindist

Given a point q and an axis-parallel rectangle r, the *mindist* of q and r, denoted as *mindist*(q, r), equals $\min_{p \in r} \|q, p\|$.



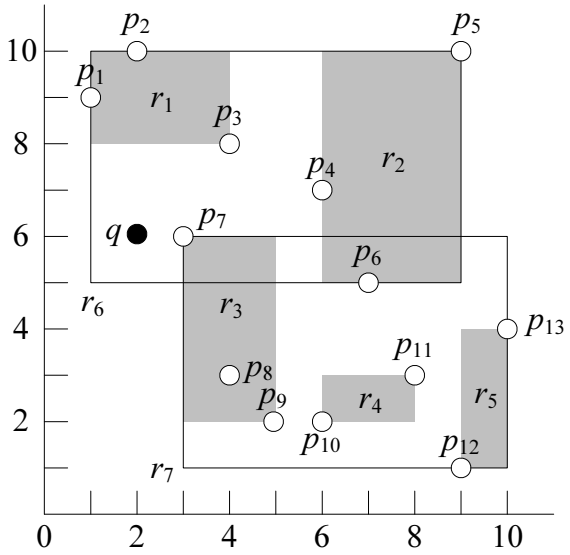
In the above example, with respect to r, the mindists of p_1 and p_2 are equal to the lengths of the two segments shown, while that of p_3 is 0.

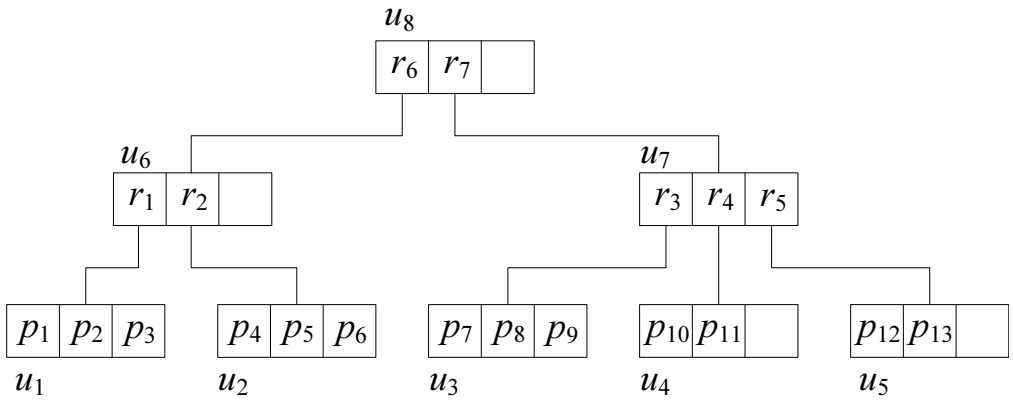
Think: how to compute *mindist*(q, r) in $O(d)$ time?

Algorithm 1: Branch-and-bound (BaB)

BaB performs a depth-first traversal of the R-tree but uses mindists to (i) prioritize the nodes for accessing, and (ii) prune the nodes that cannot contain the final answer.

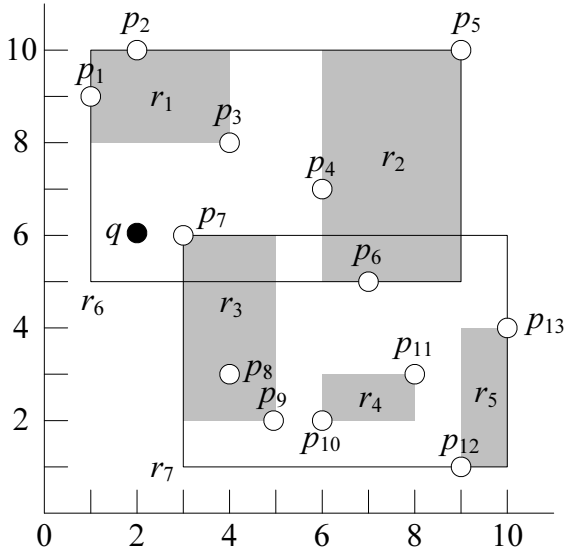
Let us illustrate the algorithm from an example. To find the NN of q (as shown in the figure), BaB starts from the root of the R-tree, where it sees two MBRs r_6 and r_7. The mindists from q to r_6 and r_7 are 0 and 1, respectively. Since $mindist(q, r_6)$ is smaller, algorithm visits u_6 next.

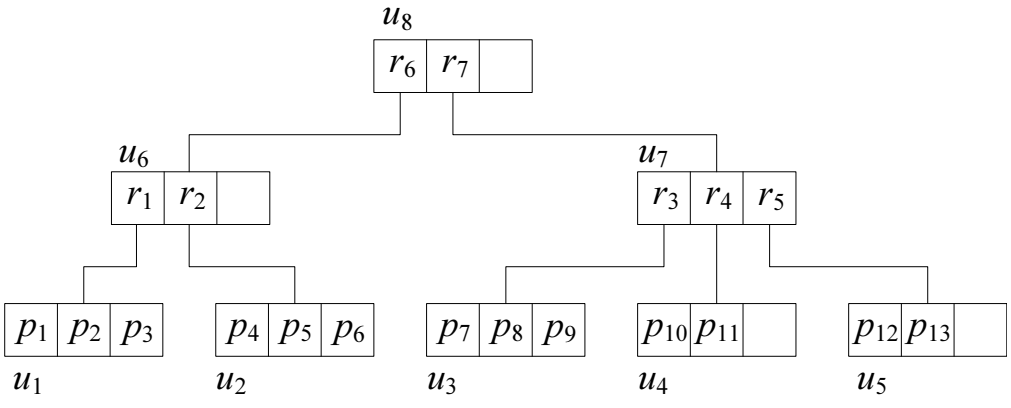




Branch-and-bound (BaB)

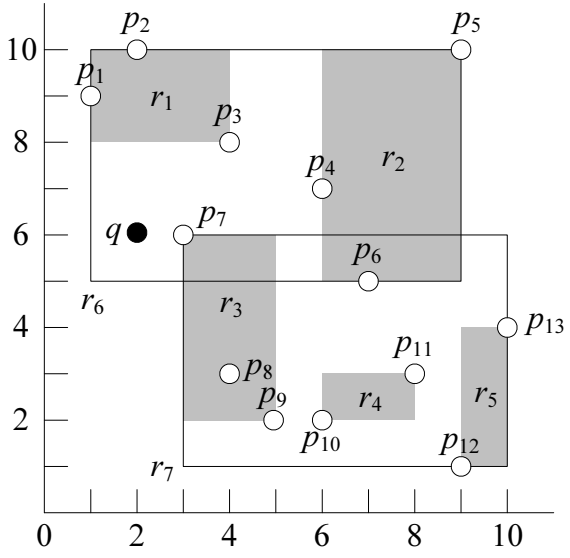
At node u_6, BaB chooses to descend into MBR r_1, because its mindist from q is smaller than that of r_2.

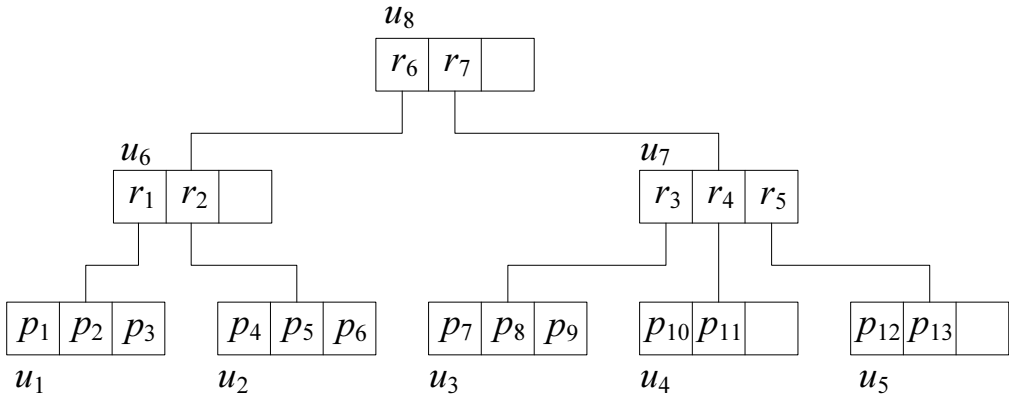




Branch-and-bound (BaB)

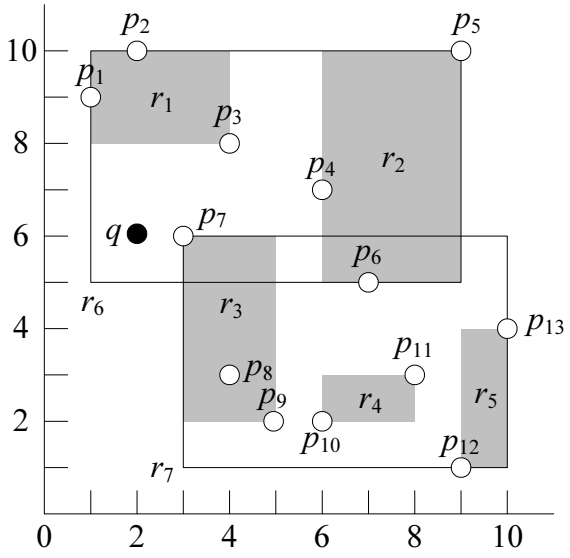
Now the algorithm is at the leaf node u_1. It simply computes the distance from q to each data point in u_1, and remembers the nearest one, i.e., p_3. This is the current NN of q found so far.

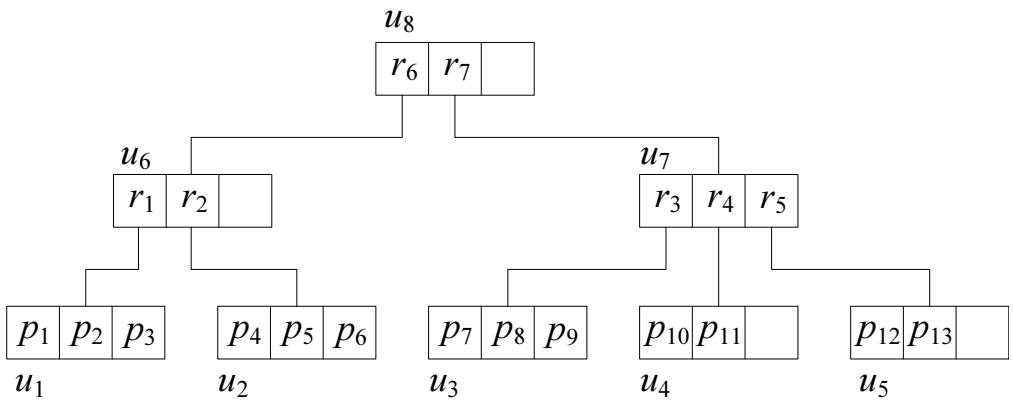




Branch-and-bound (BaB)

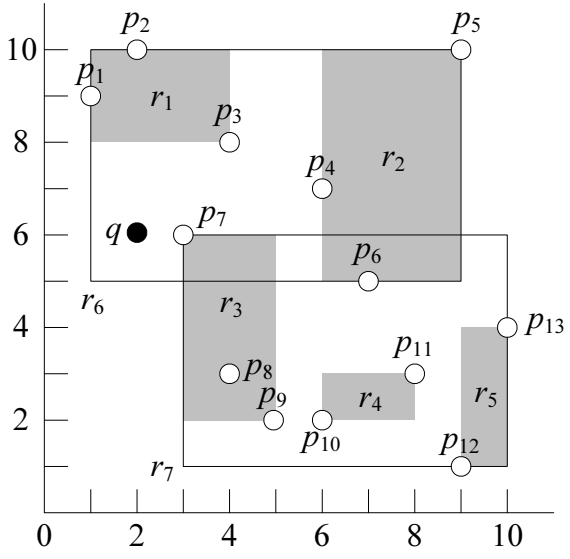
Now the algorithm backtracks to node u_6, where the subtree of MBR r_2 has not been explored yet. However, the fact that the $mindist(q, r_2) = 4$ is greater than the distance $2\sqrt{2}$ from q to the current NN p_3 rules out the possibility that the NN of q can be inside r_2. Therefore, the subtree of r_2 can be pruned.

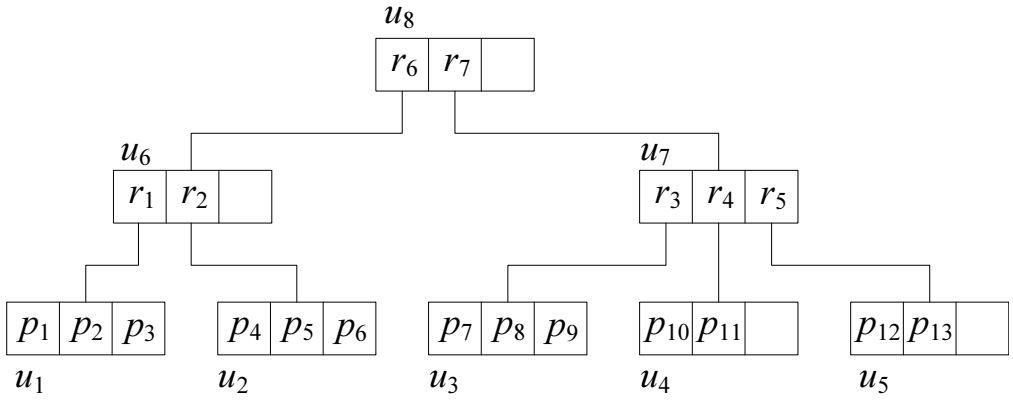




Branch-and-bound (BaB)

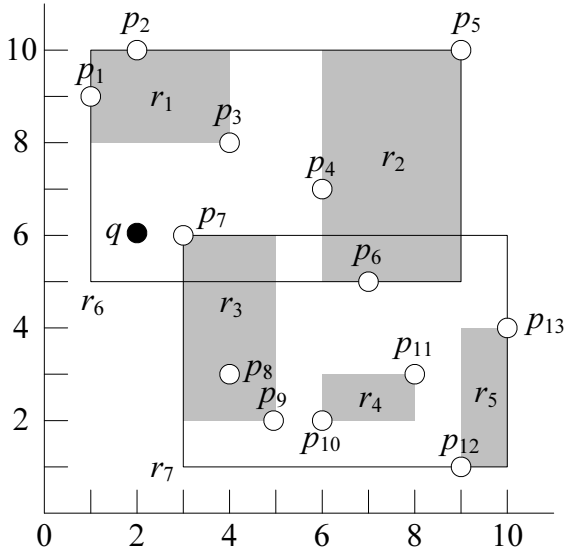
Now we backtrack to the root, where MBR r_7 has not been processed yet. The mindist 1 between q and r_7 is smaller than $\|q, p_3\| = 2\sqrt{2}$. Therefore, the child u_7 of r_7 must be visited.

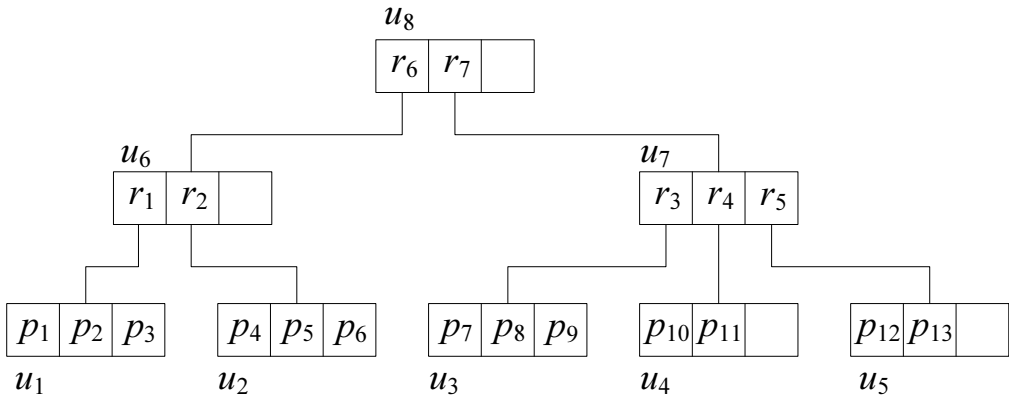




Branch-and-bound (BaB)

At node u_7, the algorithm accesses the child node u_3 of MBR r_3 which has the smallest mindist to q among r_3, r_4, r_5.

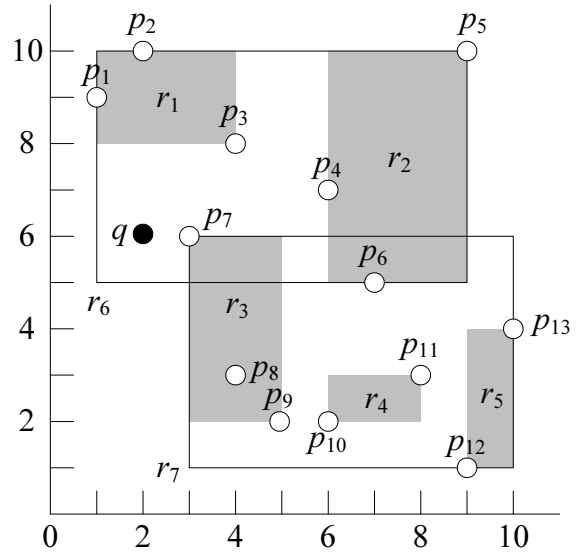


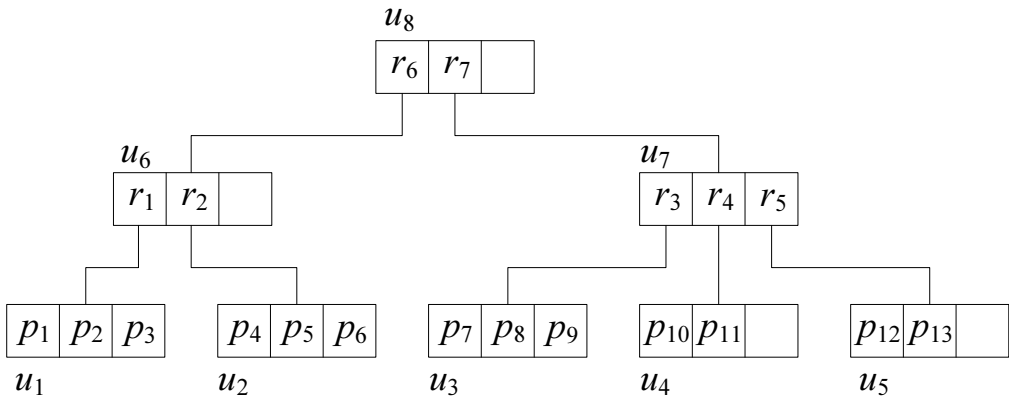


Branch-and-bound (BaB)

At node u_3, BaB finds p_7 which replaces p_3 as its current NN.

Then, it backtracks to node u_7 and prunes r_4 and r_5. After that, the algorithm backtracks one more level to the root. As all the MBRs of the root have been processed, it terminates with p_7 as the final result.





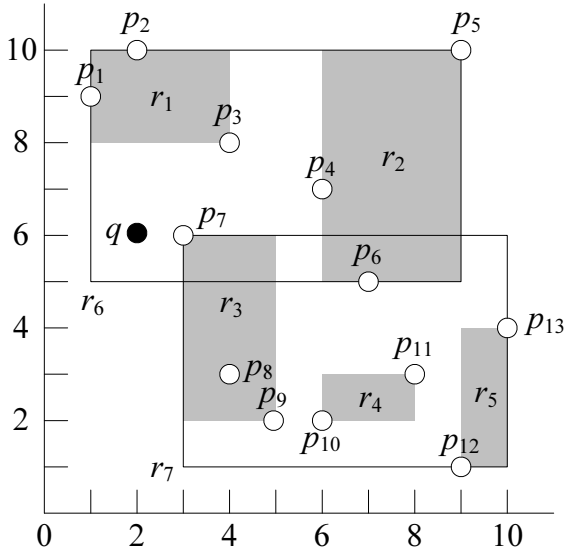
Pseudocode of BaB

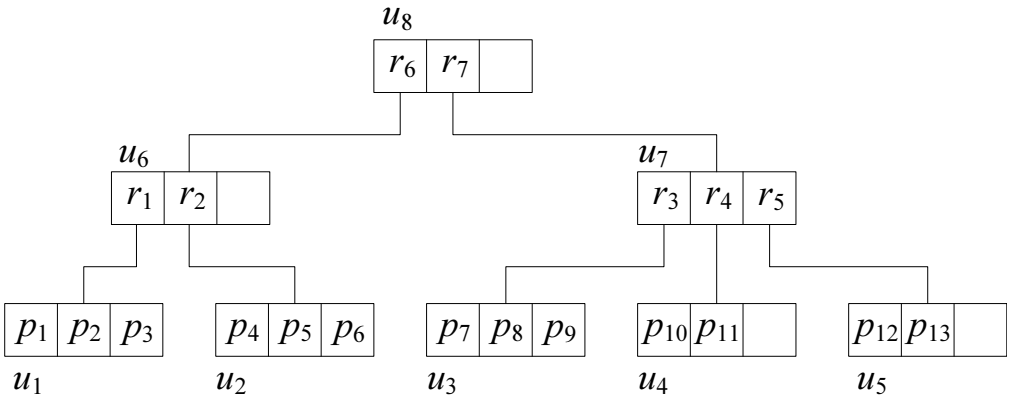
algorithm BaB(u, q)
/* u is the node being accessed, q is the query point;
p_{best} is a global variable that keeps the NN found so far;
the algorithm should be invoked by setting u to the root */

1. **if** u is a leaf node **then**
2. **if** the NN of q in u is closer to q than p_{best} **then**
3. p_{best} = the NN of q in u
4. **else**
5. sort the MBRs in u in ascending order of their mindists to q
/* let $r_1, ..., r_f$ be the sorted order */
6. **for** $i = 1$ to f
7. **if** $mindist(q, r_i) < \|q, p_{best}\|$ **then**
8. Bab(u_i, q)
/* u_i is child node of r_i */

Algorithm 2: Best First (BF)

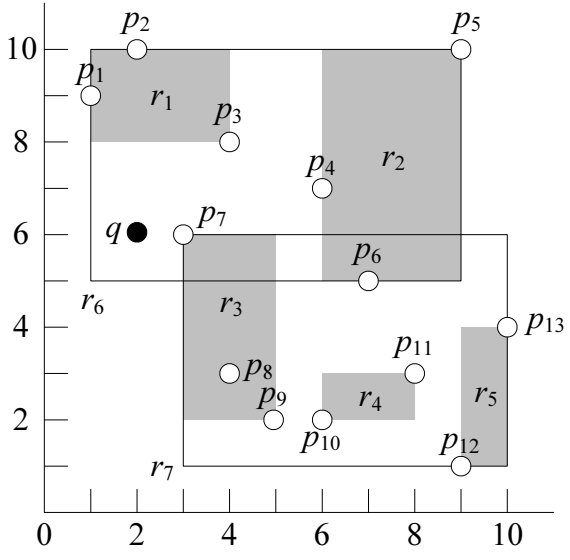
We have seen that BaB accessed u_8, u_6, u_1, u_7, u_3. Next, we will learn a better algorithm called best first (BF) that can avoid accessing u_1.

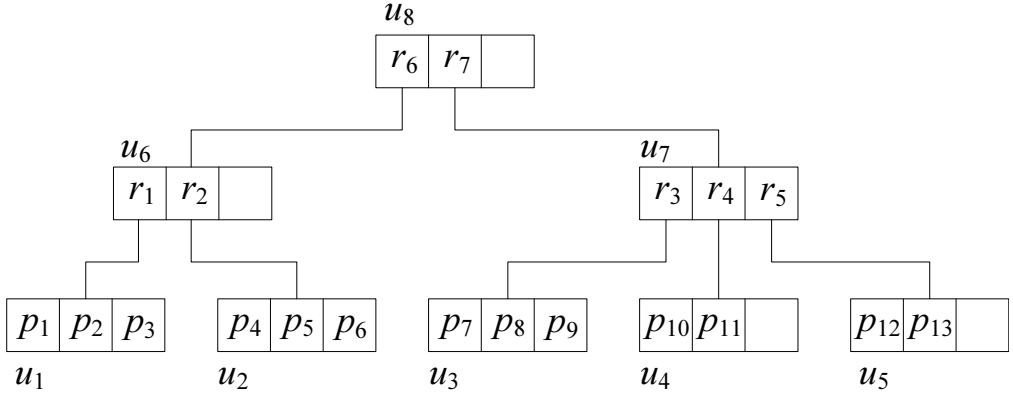




Algorithm 2: Best First (BF)

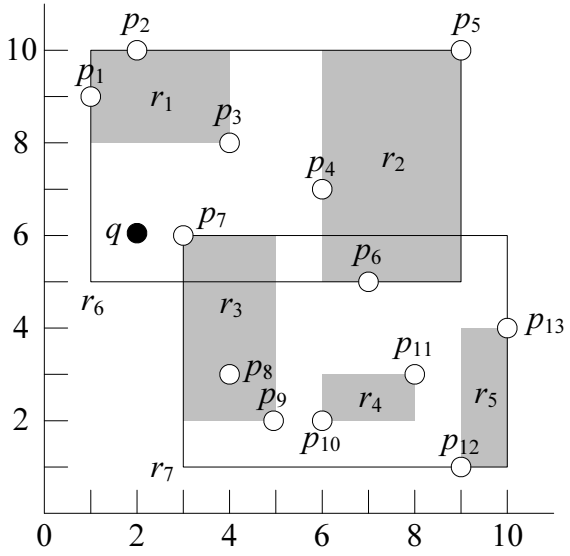
Again, we illustrate the BF algorithm with an example. As with BaB, BF also starts from the root. At any moment, the algorithm keeps in memory all the intermediate MBRs that have been seen but not yet accessed in a sorted list H, using their mindists to q as the sorting keys. In our example, so far we have seen only two MBRs r_6, r_7, so H has two entries $\{(r_6, 0), (r_7, 1)\}$.

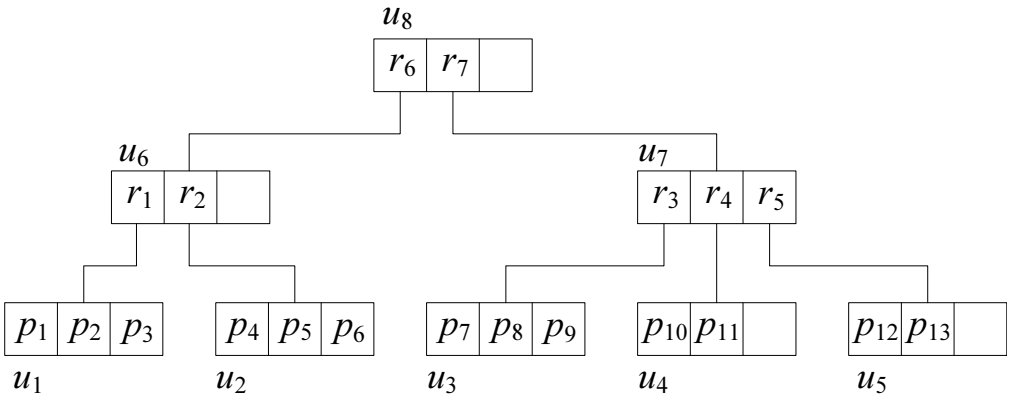




Best First (BF)

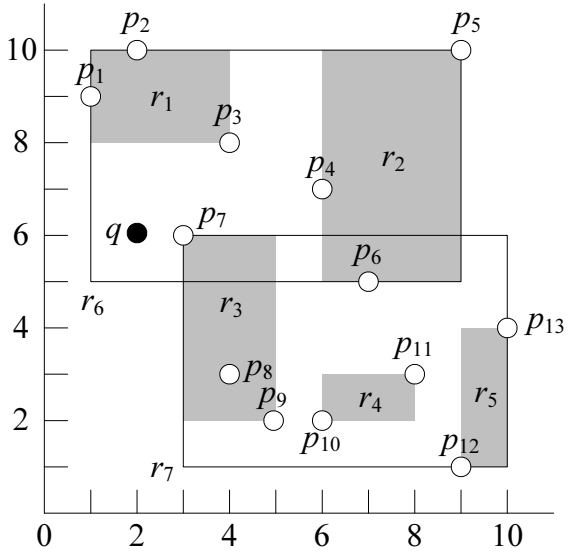
Each iteration of BF removes from H the MBR with the smallest mindist, and accesses its child node. Continuing the example, BF removes r_6 from H, visits its child node u_6, and adds to H the MBRs r_1, r_2 there. At this time, $H = \{(r_7, 1), (r_1, 2), (r_2, 4)\}$.

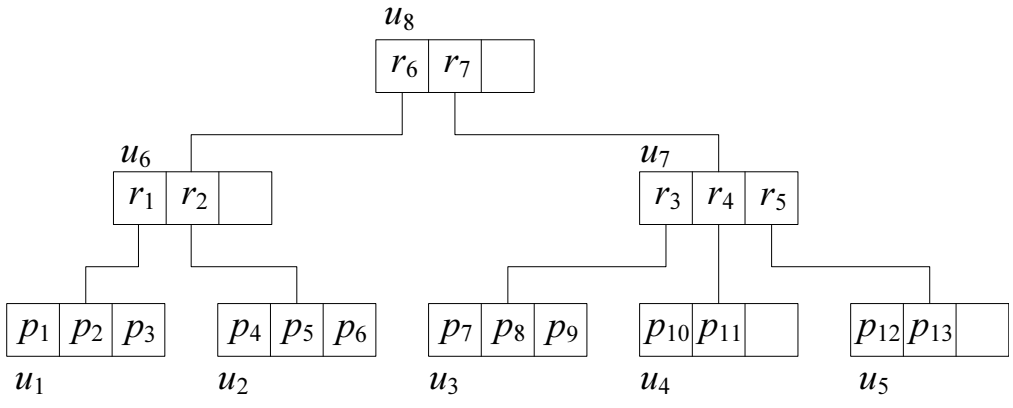




Best First (BF)

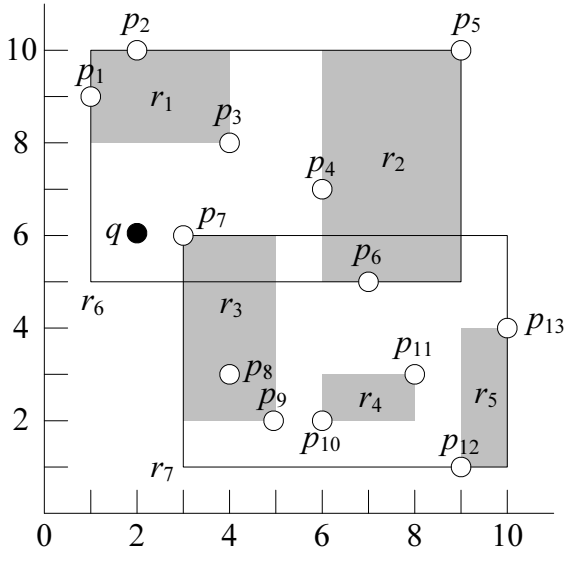
Similarly, as r_7 has the smallest key in H, BF accesses its child node u_7, after which $H = \{(r_3, 1), (r_1, 2), (r_2, 4), (r_4, 5), (r_5, \sqrt{53})\}$.

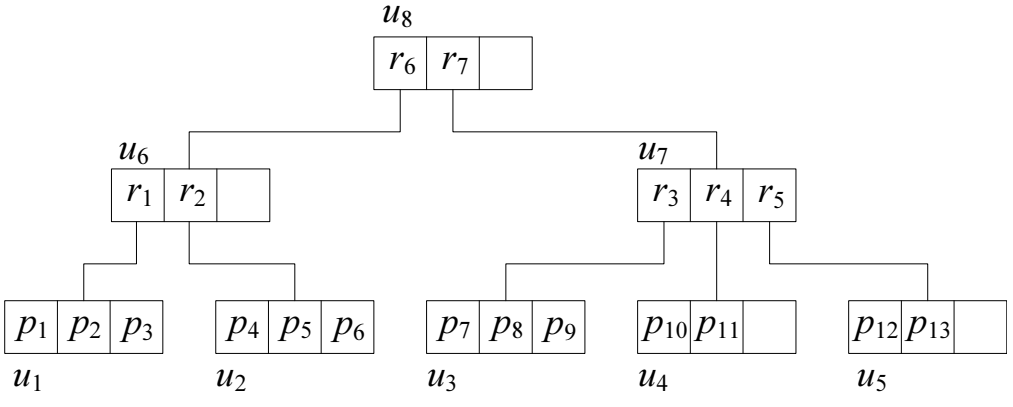




Best First (BF)

Next, the algorithm visits leaf node u_3, where p_7 is taken as the current NN. Then, BF terminates because $\|q, p_7\| = 1$ is smaller than the lowest mindist of the MBRs in $H = \{(r_1, 2), (r_2, 4), (r_4, 5), (r_5, \sqrt{53})\}$, implying that p_7 must be the final NN.





Pseudocode of BF

algorithm BF(q)
/* in the following H is a sorted list where each entry is an MBR
whose sorting key in H is its mindist to q;
p_{best} is a global variable that keeps the NN found so far. */
1. insert the MBR of the root in H
2. **while** $\|q, p_{best}\|$ is greater than the smallest mindist in H
/* if $p_{best} = \emptyset$, $\|q, p_{best}\| = \infty$ */
3. remove from H the MBR r with the smallest mindist
4. access the child node u of r
5. **if** u is an intermediate node **then**
6. insert all the MBRs in u into H
7. **else**
8. **if** the NN of q in u is closer to q than p_{best} **then**
9. p_{best} = the NN of q in u

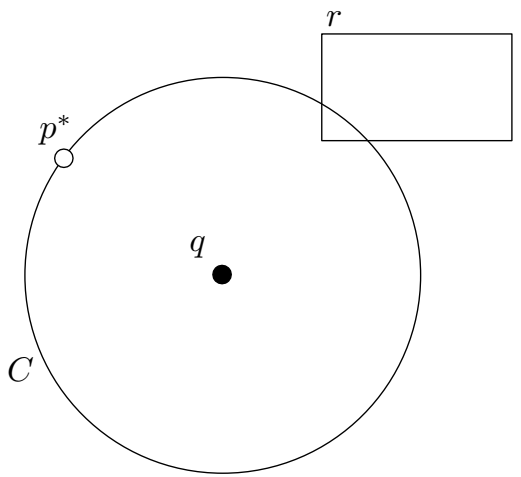
Think: what data structure would you use to manage H?

We have seen from the above examples that BF accesses less nodes than BaB. It is natural to wonder: can BF be further improved? The answer turns out to be no. As will proved next, BF is optimal, i.e., it is guaranteed to access the least number of nodes among all the algorithms that use the same R-tree to solve a given NN query.

Optimality of BF

Denote by C the circle that centers at q, and has radius $\|p^*, q\|$, where p^* is an arbitrary NN of q. Let S^* be all the nodes whose MBRs intersect C.

It is important to observe that all algorithms must access all the nodes in S^*. Assume, for example, that the node with MBR r in the figure below was not accessed. How could the algorithm assert that no point in r is closer to q than p^*?



Optimality of BF

It suffices to prove that BF accesses only those nodes whose MBRs intersect C. This can be shown in two steps:

1. BF accesses MBRs in non-descending order of their mindists to q.
   - Let r_1 and r_2 be two MBRs accessed consecutively. r_2 either already existed in H when r_1 was visited, or r_2 is an MBR inside r_1. In either case, it must hold that $mindist(q, r_2) \geq mindist(q, r_1)$.
2. Let r be the MBR of a leaf node containing an arbitrary NN of q. Let r' be an MBR that does not intersect C. By the first bullet, r is visited before r'. However, when r is found, BF must necessarily discover p^*, whose presence prevents the algorithm from accessing r' (Line 2 in Slide 21).

So far we have assumed that, if multiple data points have the smallest mindist to q, all of them must be reported.

There is an alternative version of NN search where it suffices to report one arbitrary NN in the aforementioned scenario. The BF algorithm (executed precisely as described in Slide 21) is not optimal in such a case. Can you construct a counter-example?

Extensions

BF can be adapted to solve more complicated forms of nearest neighbor search:

- Other distance metrics: So far we have assumed that the distance between two points are computed by Euclidean distance, which is known as the L_2 norm. In general, the distance between two points p and q under L_t norm—where t is an arbitrary positive value—is calculated as:

$$\left(\sum_{i=1}^{d} \left| p[i] - q[i] \right|^t \right)^{1/t}.$$

  The NN problem extends in a straightforward manner to these distance metrics (and many others).

- k nearest neighbor search: Given a query point q, return the data points with the smallest, 2nd smallest, ..., k-th smallest distances to q.

- Distance browsing: This operation outputs the points of the dataset P in ascending order of their distances to q.

Approximate Nearest Neighbor Search in High Dimensional Space

Nearest Neighbor Search

Let P be a set of n d-dimensional points in $\mathbb{R}^d$. Denote the Euclidean distance between two points $p, q \in \mathbb{R}^d$ by $\|p, q\|$.

Recall that:

> Given a query point q, a nearest neighbor (NN) query returns all the points $p \in P$ such that $\|p, q\| \leq \|p', q\|$ for $\forall p' \in P$.

In this class, the dimensionality d cannot be regarded as a constant. The dependence on d in all the complexities must be made explicit.

The Curse of Dimensionality

Many efficient nearest neighbor algorithms are known for the case when the dimensionality d is "low". However, for all the existing solutions, either the space or query time is exponential in the dimensionality d.

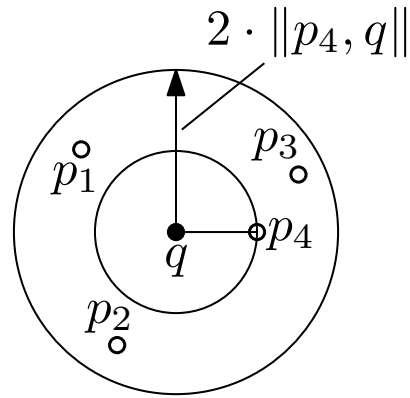
This phenomenon is called the curse of dimensionality.

One approach to deflate the curse is to trade precision for efficiency: specifically, how to achieve polynomial (in both d and n) space and query cost by accepting slightly worse neighbor points.

c-Approximate Nearest Neighbor Search

For $c > 1$, a c-approximate nearest neighbor (c-ANN) query specifies a point q. If p^* is the NN of q, the query returns an arbitrary point $p \in P$ such that $\|p, q\| \leq c \cdot \|p^*, q\|$.

- p_4 is the NN of q.
- $p_1, \ldots, p_4$ are all 2-ANNs of q.
- Any of $p_1, \ldots, p_4$ is a legal answer to the 2-ANN query w.r.t. q.



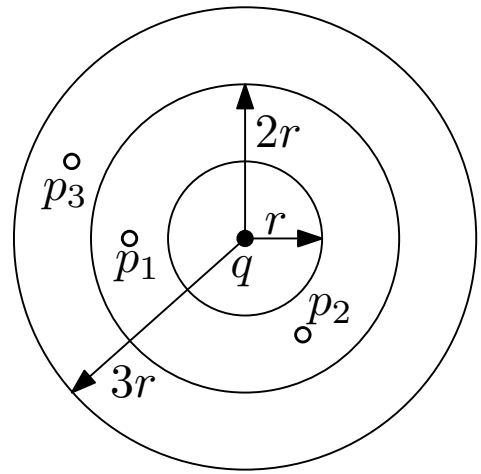
(r, c)-Near Neighbor Search

Given a point q, define $B(q, r)$ as the set of the points in P whose distances to q are at most r.

For $c > 1$, the result of an (r, c)-near neighbor query with a point q is defined as follows:

- If there exists a point in $B(q, r)$, the result must be a point in $B(q, c \cdot r)$.
- Otherwise, the result is either empty or a point in $B(q, c \cdot r)$.

- For the $(r, 2)$-near neighbor query with q, the result can be either empty or any one of p_1 and p_2.
- The result must be one of p_1, p_2 and p_3 for the $(2r, \frac{3}{2})$-near neighbor query with q.



Reduction from 4-ANN to $(r, 2)$-Near Neighbor Search

Next we show how to answer a 4-ANN query by solving a sequence of $(r, 2)$-near neighbor queries with different r values.

Remark. Our technique can be extended to reduce a $((1+\epsilon) \cdot c)$-ANN query to a sequence of (r, c)-near neighbor queries, for any value of $c > 1$ and an arbitrary constant $\epsilon > 0$.

For simplicity, let us make a mild assumption:

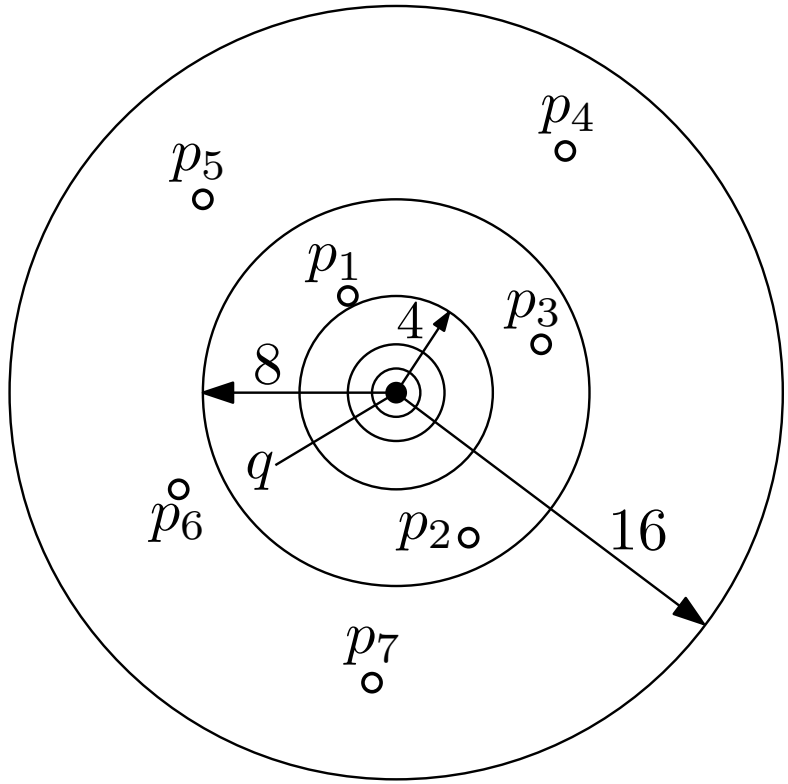
- All the point coordinates are in an integer domain of range $[1, M]$. In other words, the data space is $[1, M]^d$.

Thus, the distance between any two **distinct** points in the data space is in $[1, d_{max}]$, where $d_{max} = \sqrt{d} \cdot M$.

Reduction from 4-ANN to $(r, 2)$-Near Neighbor Search

In the figure, the radii of the circles are $1, 2, 4, 8$ and 16, respectively. Namely, the radius grows by a factor of 2.

We perform $(2^i, 2)$-near neighbor queries in ascending order of i, until a query returns a non-empty result.



Reduction from 4-ANN to $(r, 2)$-Near Neighbor Search

The 4-ANN Query Algorithm

Set $r = 1$. Repeat the following steps:

- Perform an $(r, 2)$-near neighbor query with q. If a point p is returned from the query, then return p as a 4-ANN of q.
- Otherwise, set $r = 2 \cdot r$.

Clearly, there can be at most $\lceil \log_2 d_{max} \rceil$ iterations.

Lemma: The query algorithm correctly returns a 4-ANN of a query point q.

Proof. Let p^* be the NN of q, p the point returned by the algorithm, and r^* the value of r when the algorithm terminates.

On one hand, since r^* is the **smallest** value of r such that a point in P is returned, we have $\frac{r^*}{2} < \|p^*, q\|$. Because otherwise, a point would have been returned when $r = \frac{r^*}{2}$, which contradicts with the definition of r^*. Thus, $r^* < 2 \cdot \|p^*, q\|$.

On the other hand, as p is returned from an $(r^*, 2)$-near neighbor query, $\|p, q\| \leq 2 \cdot r^*$.

Combining the above two inequalities, $\|p, q\| < 4 \cdot \|p^*, q\|$. Therefore, p is a 4-ANN of q.

□

Next we will focus on how to answer $(r, 2)$-near neighbor queries. In particular, we will consider only $r = 1$ (this does not lose generality; why?).

We will learn a new technique called locality sensitive hashing (LSH).

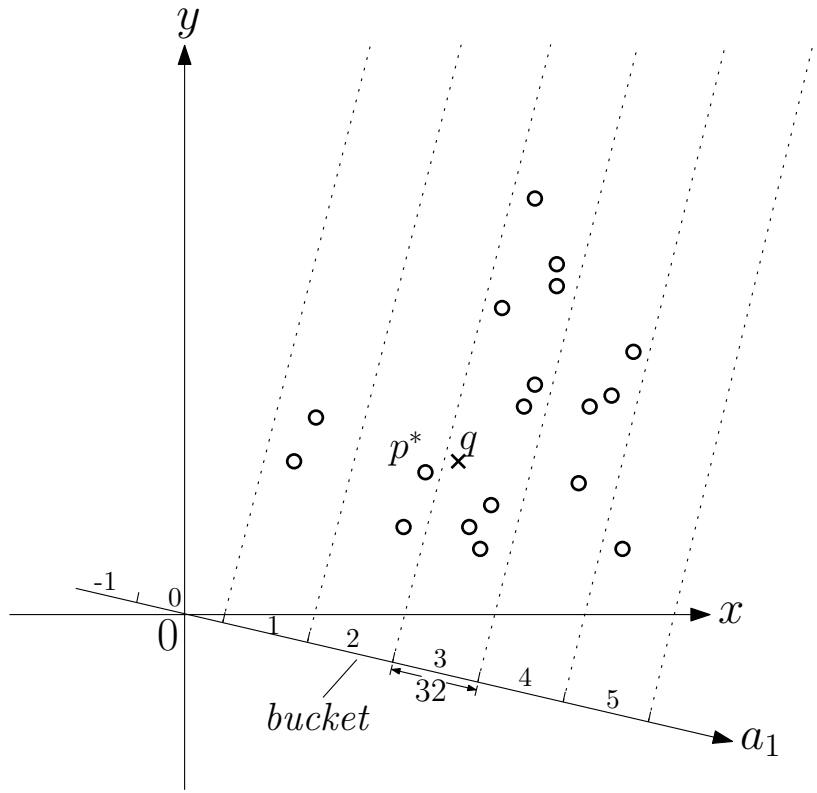
Basic Idea

First, pick a random line ℓ_1 passing through the origin. Then, chop the line into intervals of width 32. Associate each interval with a unique ID.

Let $h_1 : \mathbb{R}^d \to \mathbb{N}$ be the hash function that projects $\forall p \in \mathbb{R}^d$ into the interval with ID $h_1(p)$ of ℓ_1. As a result, each interval essentially is a hash bucket.

Observe that by h_1, "nearby" points are more likely to be hashed into the same bucket than those "far apart" points.

A hash function with such "locality preserving" property is called locality sensitive.



(p_1, p_2)-Sensitive Family

For $p_1 > p_2$, a function family $\mathcal{H} = \{h : \mathbb{R}^d \to U\}$ is called (p_1, p_2)-sensitive if for $\forall h \in \mathcal{H}$ and any two points $u, v \in \mathbb{R}^d$, we have:

- if $\|u, v\| \leq 1$, then the probability $Pr[h(u) = h(v)] \geq p_1$,
- if $\|u, v\| > 2$, then the probability $Pr[h(u) = h(v)] \leq p_2$.

There exists a (p_1, p_2)-sensitive family such that $\rho = \frac{\log 1/p_1}{\log 1/p_2} \leq 0.5$.

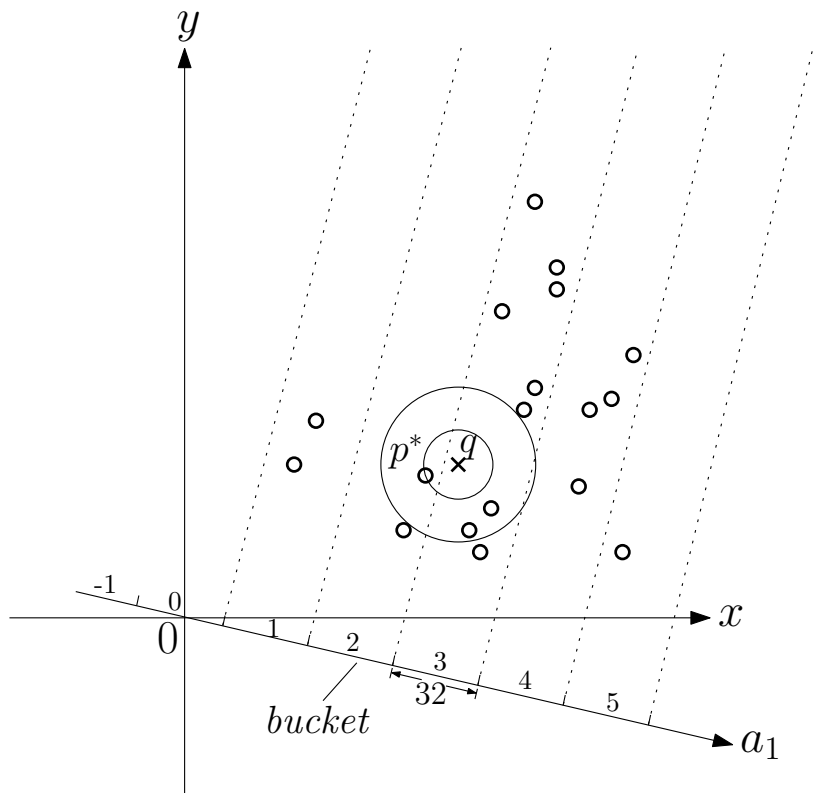
For a query point q, the points in $B(q, 1)$ are hashed into the bucket $h(q)$ with a relatively high probability. While those points that are not in $B(q, 2)$ are hashed into $h(q)$ with a smaller probability.

Intuitively, the points in the bucket $h(q)$ are more likely in $B(q, 2)$.

False Positive

For a query point q, the points u in the bucket $h(q)$ with $\|u, q\| > 2$ are called false positives.

Unfortunately, the expected number of false positives can be as large as $p_2 \cdot n$. This seriously affects the query time.



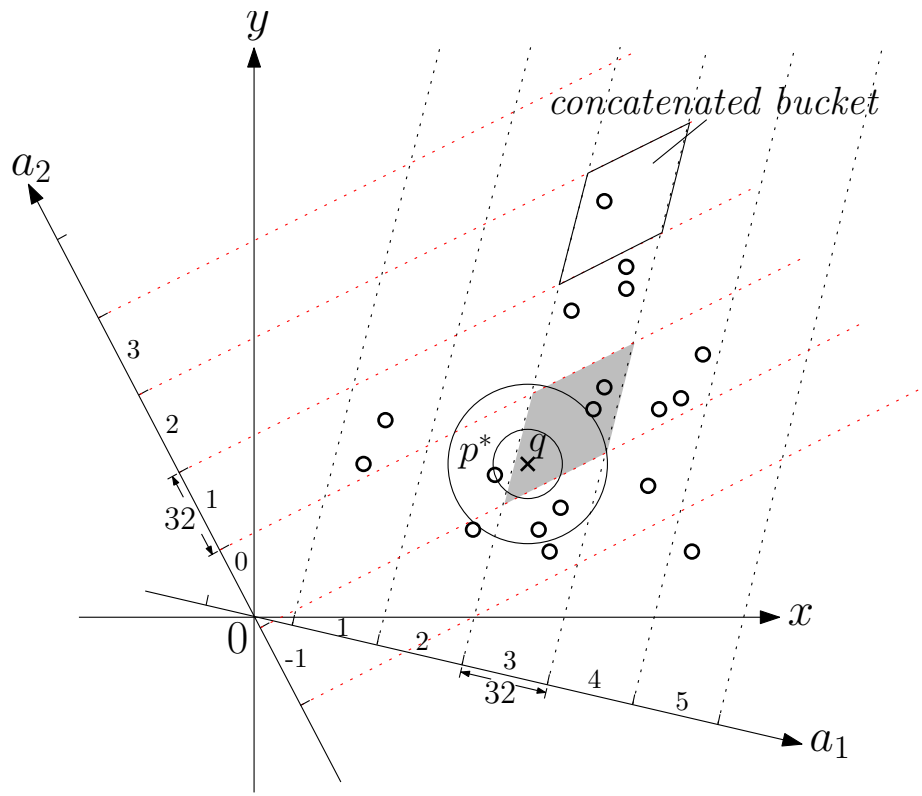
We remedy this issue by "concatenating" multiple hash functions in $\mathcal{H}$ together.

Concatenating Hash Functions

Continuing the previous example, let us generate another hash function h_2 in the same way as h_1.

Consider a hash function $g : \mathbb{R}^d \rightarrow \mathbb{N}^2$ defined by concatenating h_1 and h_2, i.e., $g(u) = (h_1(u), h_2(u))$. Each $g(u)$ corresponds to a (concatenated) bucket. $g(u) = g(v)$ **if and only if** $h_1(u) = h_1(v)$ and $h_2(u) = h_2(v)$.

As shown in the figure, the number of false positives for q in the bucket $g(q) = (3, 0)$ (i.e., the gray region) has been significantly reduced.



Concatenating Hash Functions

For an integer k, we define a function family $\mathcal{G} = \{g : \mathbb{R}^d \to U^k\}$, where each $g(u) = (h_1(u), h_2(u), \cdots, h_k(u))$ consists of k hash functions chosen independently and uniformly from an (p_1, p_2)-sensitive family $\mathcal{H}$.

For any two points $u, v \in \mathbb{R}^d$, $g(u) = g(v)$ **if and only if** $h_i(u) = h_i(v)$ for all $i = 1, \cdots, k$. Thus, $Pr[g(u) = g(v)] = \prod_{i=1}^{k} Pr[h_i(u) = h_i(v)]$. Hence:

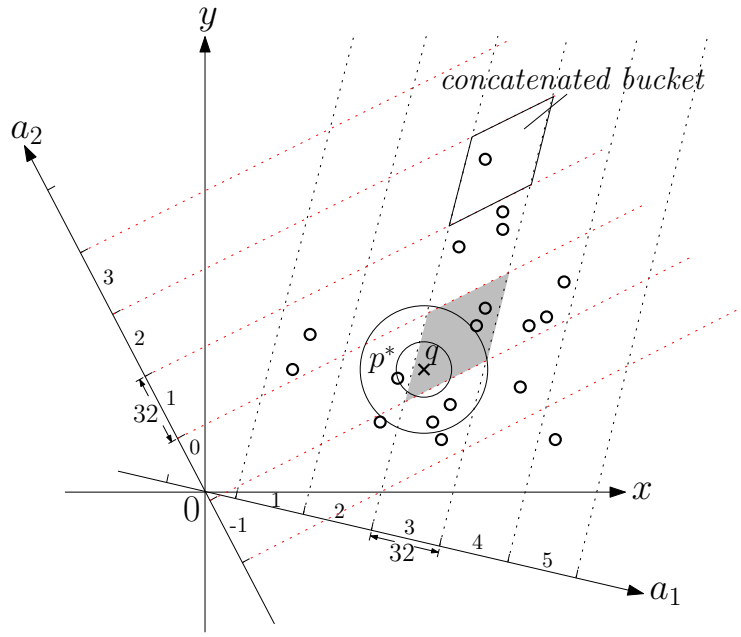
- if $\|u, v\| \leq 1$, then $Pr[g(u) = g(v)] \geq p_1^k$,
- if $\|u, v\| > 2$, then $Pr[g(u) = g(v)] \leq p_2^k$.

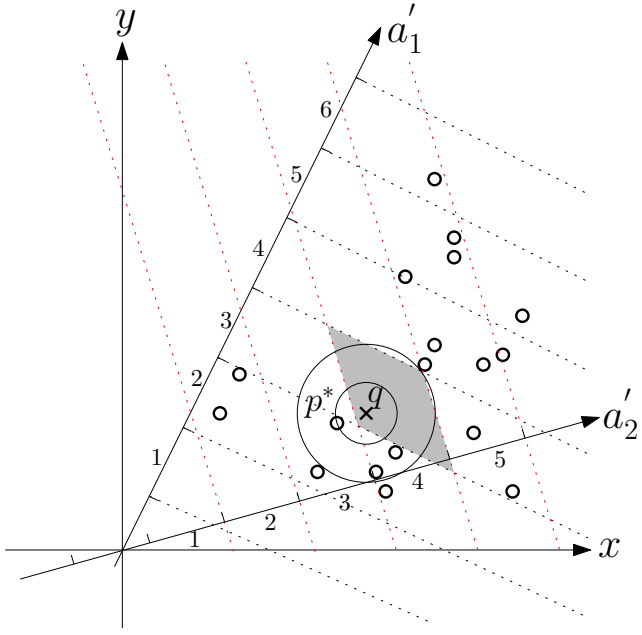
Therefore, the function family $\mathcal{G}$ is (p_1^k, p_2^k)-sensitive.

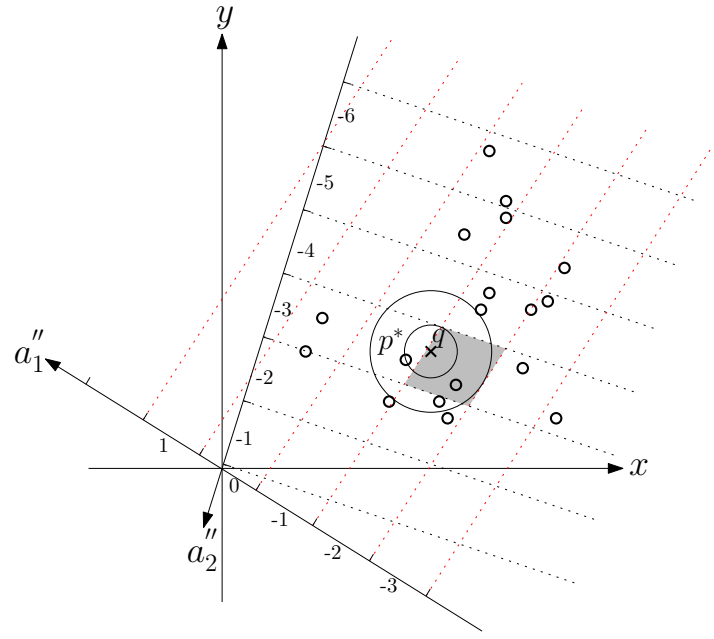
Remark. By a hash function $g \in \mathcal{G}$, the expected number of false positives is reduced to $p_2^k \cdot n$. However, in the meanwhile, the probability for a point in $B(q, 1)$ being hashed into $g(q)$ also decreases to as small as p_1^k.

The Repeating Trick

To increase the probability for a near neighbor being hashed into the same bucket of q, we repeatedly use different hash functions from $\mathcal{G}$ to construct different hash tables.







The LSH Technique

For an integer L, the LSH constructs L hash tables for P as follows:

- Independently and uniformly choose L functions $g_1, g_2, \cdots, g_L$ from the (p_1^k, p_2^k)-sensitive function family $\mathcal{G}$.
- For each g_i, construct a hash table for P by hashing each point $u \in P$ into bucket $g_i(u)$.

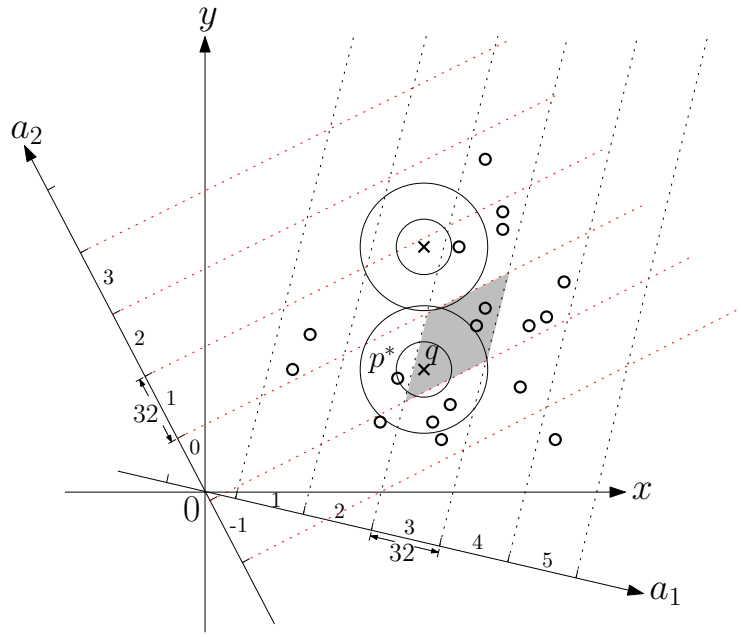
The $(1, 2)$-Near Neighbor Query Algorithm

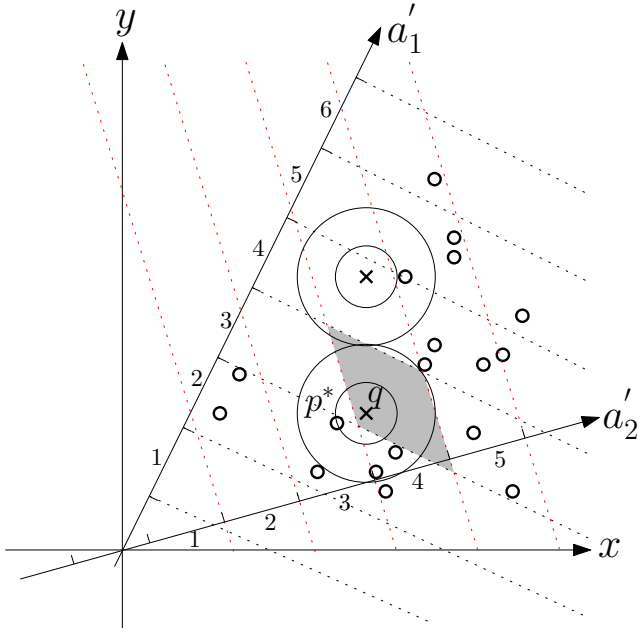
For a query point q, inspect the L hash buckets $g_1(q), \cdots, g_L(q)$ by checking each point u therein:

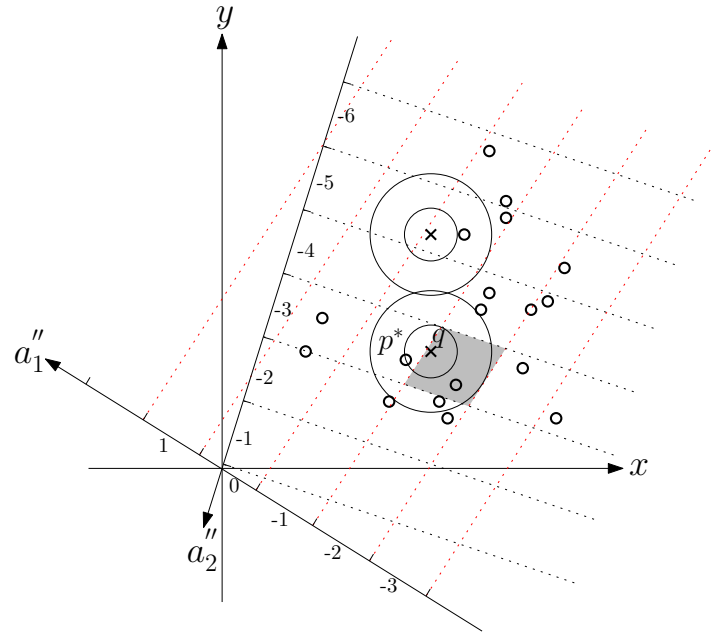
- If $\|u, q\| \leq 2$, then return u.
- Otherwise, if so far in total $3 \cdot L$ or all the points in the L buckets have been checked, then terminate and return nothing.

Query Examples

Theoretically speaking, we do need to construct a sufficiently large number of hash tables to ensure correctness. However, in most cases, about 10 hash tables are enough to answer queries. In this example, we only need three.







Correctness

For a fixed query point q, consider the following two events:

- E_1: If there exists a point $u \in B(q, 1)$, then $g_i(u) = g_i(q)$ for some $i \in \{1, 2, \cdots, L\}$.
- E_2: The total number of false positives in the L buckets $g_1(q), g_2(q), \cdots, g_L(q)$ is less than $3 \cdot L$.

Lemma: When both E_1 and E_2 hold at the same time, the query algorithm correctly answers an $(1, 2)$-near neighbor query with q.

Correctness

Proof. Let $|g_i(q)|$ be the number points in the bucket $g_i(q)$. Observe that the query algorithm examines at most $\min\{\sum_i |g_i(q)|, 3 \cdot L\}$ points.

When $\sum_i |g_i(q)| < 3 \cdot L$, by the fact that E_1 holds, if there exists $u \in B(q, 1)$, then u is in at least one of the L buckets. Thus, u must have been checked. Hence, a point in $B(q, 2)$ must be returned. On the other hand, if $B(q, 1) = \emptyset$, then either reporting a point in $B(q, 2)$ or not is correct.

When the algorithm has checked $3 \cdot L$ points, since E_2 holds, there must be at least one point in $B(q, 2)$. Hence, one such point will be returned.

□

Next, we show that:

> By setting the values of k and L carefully, both the two events E_1 and E_2 hold at the same time with at least constant probability.

In other words, the query algorithm correctly answers an $(1, 2)$-near neighbor query with q with at least constant probability.

Before we jump into the technical details, let us first get an idea of the basic direction to set k and L.

On one hand, as the expected number of false positives in $g_i(q)$ is $p_2^k \cdot n$, its total expected number over all the L buckets is $L \cdot p_2^k \cdot n$. If we can make this total expectation $\leq L$, then its actual value is not likely to be much larger than L. As a result, $L \cdot p_2^k \cdot n \leq L \Rightarrow k \geq \log_{1/p_2} n$.

On the other hand, since $Pr[g_i(u) = g_i(q)] \geq p_1^k$ for a point $u \in B(q, 1)$, the probability of $g_i(u) \neq g_i(q)$ for all the L buckets is $\leq (1 - p_1^k)^L$. We will show that this probability is no more than a constant when $L \geq 1/p_1^k$. As a result, the probability of at least one $g_i(u) = g_i(q)$ among all the L buckets is $\geq 1 - (1 - p_1^k)^L$ which is greater than a constant.

Thus, we set $k = \lceil \log_{1/p_2} n \rceil$ and $L = \lceil \frac{\sqrt{n}}{p_1} \rceil \geq \lceil \frac{n^\rho}{p_1} \rceil \geq \lceil \frac{1}{p_1^k} \rceil$ for $\rho = \frac{\log 1/p_1}{\log 1/p_2} \leq 0.5$.

In what follows, we will prove that both $Pr[E_1]$ and $Pr[E_2]$ are greater than a constant under the above values of k and L.

Preliminary 1: Markov's Inequality

For a nonnegative random integer variable X and $t > 0$, we have:

$$Pr[X \geq t] \leq \frac{E[x]}{t}.$$

Proof.

$$\begin{aligned} E[X] &= \sum_x x \cdot Pr[X = x] \\ &\geq \sum_{x \geq t} x \cdot Pr[X = x] \\ &\geq t \sum_{x \geq t} Pr[X = x] \\ &= t \cdot Pr[X \geq t] \end{aligned}$$

□

Preliminary 2:

For $x \geq 1$, $(1 - \frac{1}{x})^x \leq \frac{1}{e}$ holds.

Proof. By the well-known inequality $1 + y \leq e^y$ for $|y| \leq 1$, we have:

$$(1 - \frac{1}{x})^x \leq e^{-\frac{1}{x} \cdot x} = \frac{1}{e}$$

for $x \geq 1$.

□

Preliminary 3: Union Bound

For two events A and B, we have:

$$Pr[A \cup B] = Pr[A] + Pr[B] - Pr[A \cap B] \leq Pr[A] + Pr[B].$$

The event

- E_1: If there exists a point $u \in B(q, r)$, then $g_i(u) = g_i(q)$ for some $i \in \{1, 2, \cdots, L\}$.

holds with at least probability of $1 - \frac{1}{e}$, for $k = \lceil \log_{1/p_2} n \rceil$ and $L = \lceil \frac{\sqrt{n}}{p_1} \rceil$.

Proof. Since for a point $u \in B(q, 1)$, we have $Pr[g_i(u) = g_i(q)] \geq p_1^k$ for $\forall i = 1, \ldots, L$. Thus, $Pr[\bigwedge_{i=1}^{L} g_i(u) \neq g_i(q)] \leq (1 - p_1^k)^L$.

As $k = \lceil \log_{1/p_2} n \rceil$, we have $p_1^k \geq \frac{p_1}{n^\rho} \geq \frac{p_1}{\sqrt{n}} \geq \frac{1}{L}$. Thus,

$$Pr[\bigwedge_{i=1}^{L} g_i(u) \neq g_i(q)] \leq (1 - p_1^k)^L \leq (1 - \frac{1}{L})^L \leq \frac{1}{e}.$$

Therefore, $Pr[E_1] = 1 - Pr[\bigwedge_{i=1}^{L} g_i(u) \neq g_i(q)] \geq 1 - \frac{1}{e}$.

□

The event

- E_2: The total number of false positives in the L buckets $g_1(q), g_2(q), \ldots, g_L(q)$ is less than $3 \cdot L$.

holds with at least probability of $\frac{2}{3}$, for $k = \lceil \log_{1/p_2} n \rceil$ and $L = \lceil \frac{\sqrt{n}}{p_1} \rceil$.

Proof. The expected number of false positive in $g_i(q)$ is at most $p_2^k \cdot n \leq 1$. Denote by X the random variable of the total number of false positives over all $g_i(q)$'s. Thus, $E[X] \leq L$.

By Markov's inequality, we have $Pr[X \geq 3 \cdot L] \leq \frac{E[X]}{3 \cdot L} \leq \frac{1}{3}$. Therefore, $Pr[E_2] = 1 - Pr[X \geq 3 \cdot L] \geq \frac{2}{3}$.

□

Finally, by the Union Bound, $Pr[\bar{E}_1 \cup \bar{E}_2] \leq Pr[\bar{E}_1] + Pr[\bar{E}_2] \leq \frac{1}{e} + \frac{1}{3}$. Hence, $Pr[E_1 \cap E_2] \geq 1 - \frac{1}{e} - \frac{1}{3} = \frac{2}{3} - \frac{1}{e}$.

Therefore,

There exists a (p_1, p_2)-sensitive family such that by setting $k = \lceil \log_{1/p_2} n \rceil$ and $L = \lceil \frac{\sqrt{n}}{p_1} \rceil$, the LSH correctly answers an $(1, 2)$-near neighbor query with probability at least $\frac{2}{3} - \frac{1}{e}$.

Query Time

For a query point q, the time for computing $g_1(q), \cdots, g_L(q)$ is $O(d \cdot k \cdot L)$, and the time for checking at most $3 \cdot L$ points is $O(d \cdot L)$. Thus, the total query time is bounded by $O(d \cdot k \cdot L) = O(d \cdot \sqrt{n} \cdot \log n)$.

Space

The space consumption consists of two parts: (i) the space $O(d \cdot n)$ for storing P, and (ii) the space $O(n \cdot L) = O(n^{1.5})$ for the L hash tables. Hence, the total space consumption is $O(d \cdot n + n^{1.5})$.

Remark. The value $L = \lceil \frac{\sqrt{n}}{p_1} \rceil$ is only valid for $\rho = \frac{\log 1/p_1}{\log 1/p_2} \leq 0.5$ for some specific (p_1, p_2)-sensitive families. In fact, for any such family this bound does not always hold, in which case, we can only bound $L = \lceil \frac{n^\rho}{p_1} \rceil$.

Nevertheless, all our previous analysis applies to any (p_1, p_2)-sensitive family $\mathcal{H}$ (and hence, $\mathcal{G}$) by using $L = \lceil \frac{n^\rho}{p_1} \rceil$. In other words, both query time and space consumption essentially depend on the value of ρ.

Different families $\mathcal{H}$ have various ρ values, and hence would result in different performance. The smaller value of ρ the better performance can be achieved.

A (p_1, p_2)-Sensitive Family

A well-known (p_1, p_2)-sensitive family $\mathcal{H} = \{h : \mathbb{R}^d \to \mathbb{N}\}$ with $\rho \leq 0.5$ for the Euclidean distance has the following form:

$$h(u) = \lfloor \frac{\vec{a} \cdot \vec{u} + b}{w} \rfloor,$$

where:

- $\vec{a}$ is a d-dimensional vector, whose each coordinate is chosen independently from the standard Gaussian Distribution $N(0, 1)$;
- w is an appropriate integer (e.g., $w = 32$); and
- b is a real value uniformly drawn from the range $[0, w)$.