On the Generalization of Nearest Neighbor Queries

29 October 2001

Abstract

Nearest neighbor queries on R-trees use a number of pruning techniques to improve the search. We examine three common 1-nearest neighbor pruning strategies and generalize them to k-nearest neighbors. This generalization clears up a number of prior misconceptions. Specifically, we show that the generalization of one pruning technique, referred to as strategy 2, is non-trivial and requires the introduction of a new algorithm we call *promise-pruning*. In addition, we show that, contrary to other claims, applying this generalized strategy to k-nearest neighbor queries results in a theoretically better search. This discovery is reinforced with empirical results showing the success of promise-pruning on both random and real-world data.

1 Introduction

This paper clarifies some misconceptions in the literature related to pruning strategies in nearest neighbor queries on R-trees. We show that the 1-nearest neighbor (1-nn) algorithm in (Roussopoulos et al., 1995) does *not easily generalize* to k-nearest neighbors. We review definitions of the three 1-nearest neighbor pruning strategies used in the algorithm, abstract them to k-nearest neighbors (k-nn) and demonstrate their correctness. To the best of our knowledge, we are the first to explicitly provide these general definitions. Moreover these generalizations lead to interesting results that contradict some previous claims.

First, we show that blindly applying the second pruning technique (commonly called strategy 2) to k-nn queries may result in incorrect neighbor findings. Second we introduce an algorithm called *promise-pruning* that generalizes strategy 2 to k-nn queries but *does* return accurate nearest neighbor information. Third, we show promise-pruning leads to a theoretically better search. That is, we show that strategy 2, when generalized, is applicable to nearest neighbor queries where $k > 1$ and provably improves the query. This contradicts prior claims in (Cheung & Fu, 1998) that strategy 2 cannot help the search and (Hjaltason & Samet, 1999) that strategy 2 is only applicable when $k = 1$.

2 R-trees

R-trees and their variants (Guttman, 1984; Sellis et al., 1988; Beckmann et al., 1990) are data structures for storing and retrieving spatial objects in Euclidean space. They support dynamic insertion and deletion methods and are frequently used for nearest neighbor queries. Most nearest neighbor queries use pruning techniques to improve the search. Like most trees, R-trees are decomposed into internal and leaf nodes. Both types of nodes contain records, but records of internal nodes differ slightly from those of leaf nodes. A record r, belonging to an internal node, is a tuple $\langle M, \mu \rangle$ where μ is a pointer to the child node of r and M is an n-dimensional minimum bounding rectangle (MBR). M tightly bounds the spatial objects located in the subtree of r. For example, given the points $(1, 2)$ and $(4, 5)$, the MBR would be $\langle (1, 4), (2, 5) \rangle$. The records of leaf nodes are also tuples but have the form $\langle M, o \rangle$ where o is either the literal spatial object or an identification to it. M is the minimum bounding rectangle of o. For our purposes, we also assume that spatial objects are unique to a particular R-tree. That is, no two leaf node records point to the same spatial object in memory.

The number of records in a node is its branching factor. Every node of an R-tree contains between b and B records where $b \leq \lfloor \frac{B}{2} \rfloor$. One exception is the root node, which must have at least two records. R-trees are completely balanced, meaning all leaf nodes have the same depth.

For explanatory purposes, we often refer to the minimum bounding rectangle of a node instead of the minimum bounding rectangle of the record pointing to that node. Similarly, when we refer to the children of a node, we mean the children pointed to by the records of the node.

Since every non-root node is tightly enclosed by a minimum bounding rectangle, meaningful spatial relationships may exist between nodes. It is these relationships that are exploited when using R-trees for nearest neighbor queries.

3 Metrics

Generally, nearest neighbor queries are of the form: *Find the k nearest items with respect to some other item*. For R-trees, nearest neighbor queries are of the form: *Given a query point q and an R-tree T with spatial objects of matching dimension to q, find the k-nearest objects to q in T.* Nearest here means Euclidean distance. Throughout the rest of the paper, the term *distance* means *Euclidean distance*.

Roussopoulos et al. (1995) define the 1-nearest neighbor algorithm given in Figure 1. This procedure is a depth-first search through the R-tree using three pruning strategies based on two metrics that provide lower and upper bound information about a node's children.

```
1-NN(query, rtree)
1.  nearest.dist ← ∞
2.  nearest.obj ← nil
3.  1-NNSEARCH(query, rtree.root, nearest)
4.  RETURN(nearest)

1-NNSEARCH(query, node, nearest)
1.  IF node.type = leaf
2.      FOREACH record in node.records
3.          WHEN DISTANCE(query, record.object) < nearest.dist
4.              nearest.dist ← DISTANCE(query, record.object)
5.              nearest.obj ← record.object
6.  ELSE
7.      activeBranchList ← node.records
8.      sort activeBranchList by MinDistance
9.      STRATEGYONE(query,activeBranchList)
10.     STRATEGYTWO(query,activeBranchList, nearest)
11.     FOREACH record in activeBranchList
12.         1-NNSEARCH(query, record.child, nearest)
13.         STRATEGYTHREE(query,activeBranchList, nearest.dist)
```

Figure 1: The original 1-nearest neighbor algorithm presented in (Roussopoulos et al., 1995)

The first metric, $MinDistance(q, M)$, is the length of the shortest line between the query point q and the nearest face of the MBR M. When the query point lies within the MBR, $MinDistance$ is 0. Figure 2 shows the $MinDistance$ values for a query point and three minimum bounding rectangles. Because an MBR tightly encapsulates the spatial objects within it, each face of the MBR must touch at least one of the objects it encloses (Roussopoulos et al., 1995). This is called the MBR face property.

From Figure 2 it is apparent that $MinDistance$ is a lower bound, or optimistic estimate of the distance between a query point and a spatial object located inside an MBR. The actual distance between a query point and the closest object in the MBR however, may be much larger than the $MinDistance$ estimate.

The second metric defined by Roussopoulos et al. (1995) is called $MinMaxDistance$. It is meant as an upper bound, or pessimistic estimate of the distance between a query point and a spatial object located within a minimum bounding rectangle. Figure 2 depicts these distances for three MBRs and a query point. From its name one can see $MinMaxDistance$ is calculated by finding the minimal distance from a set of maximal distances.

This set of maximal distances is formed as follows: Suppose we have an n-dimensional minimum bounding rectangle. If we fix one of the dimensions, we are left with two $n - 1$ dimensional hyperplanes;

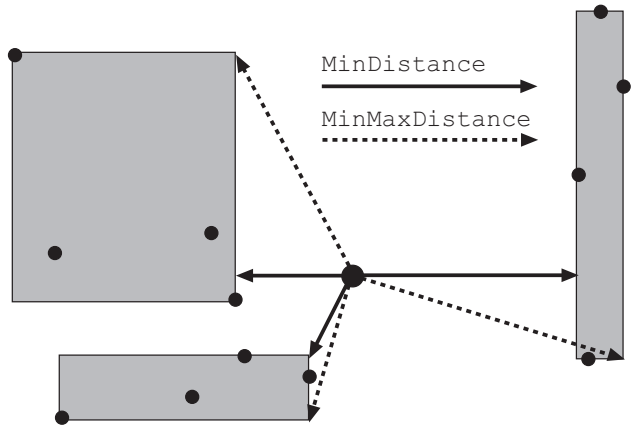


Figure 2: A visual explanation of $MinDistance$ and $MinMaxDistance$ in two dimensions.

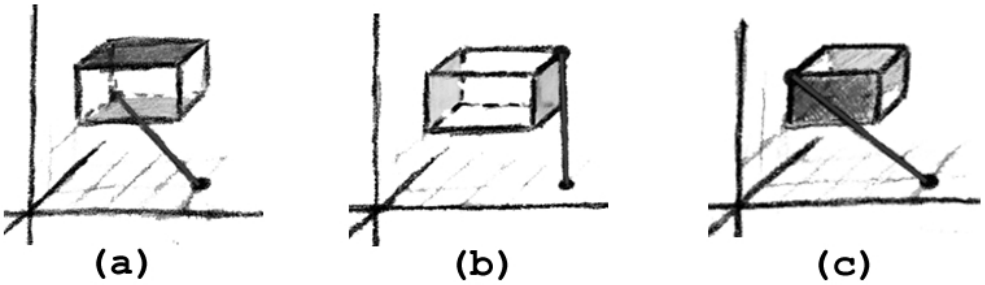


Figure 3: A visual explanation of $MinMaxDistance$ in three dimensions. Each picture depicts the two planes of the minimum bounding rectangle formed when fixing (a) the y-axis; (b) the x-axis; and (c) the z-axis.

one representing the MBR's lower bounds, the other representing its upper bounds. We know from the MBR face property that at least one spatial object touches each of these hyperplanes. However, given only the MBR, we cannot identify this location. But, given a query point, we can say that an object is at least close as the distance from that point to the farthest point on the closest hyperplane. This distance is clearly an upper bound on the distance between the query point and a spatial object located within the MBR. By iteratively fixing each dimension of an MBR and finding the upper bound, we can form the set of maximal distances. Since each maximal distance is an upper bound, it follows that the minimum of these is also an upper bound. This minimum distance is what we call the $MinMaxDistance(q, M)$ of a query point q and an MBR M.

Figure 3 provides a visual example of finding the $MinMaxDistance$ for a query point and MBR in three dimensions. When fixing a single dimension in 3-space, we end up with 3 pairs of planes. Each constituent of Figure 3 shows first, how the closest plane to the query point is chosen, and second, how the farthest point on that plane results in a maximum distance. The $MinMaxDistance$ then, is the shortest of

the three lines (i.e., (b)).

Using these metrics, one can construct pruning strategies. These pruning strategies can be applied to branch and bound style searches to yield effective nearest neighbor queries on R-trees.

3.1 Pruning Strategy 1

Pruning strategy 1 removes minimum bounding rectangles from the *activeBranchList* without using the current nearest neighbor estimate. For this reason, we call strategy 1 *self-pruning*. Self-pruning works by comparing an MBR's $MinMaxDistance$ against another MBR's $MinDistance$. If the $MinDistance$ for an MBR is greater than the $MinMaxDistance$ of another MBR then it is discarded because the MBR is guaranteed to bound a closer spatial object. A formal definition of this pruning strategy is given below.

Definition 1 (Strategy 1 for 1-nn). *Given a query point q and a list of minimum bounding rectangles $\mathcal{M}$, discard any minimum bounding rectangle $M_i \in \mathcal{M}$ if there exists $M_j \in \mathcal{M}$ such that $MinDistance(q, M_i) > MinMaxDistance(q, M_j)$*

The intuition behind self-pruning is that examining an MBR M_i whose lower bound estimate is greater than the upper bound estimate of another MBR M_j is useless. This is clear in the 1-nearest neighbor case since M_j is guarenteed to contain an object closer to the query point than M_i. However, with k-nearest neighbors, we may still be interested in M_i since M_j may not contain the k-nearest objects. By definition of $MinMaxDistance$, we can only guarantee that M_j contains *one* spatial object with distance closer to the query point than M_i. So, in order to prune an MBR M_i with strategy 1 for k-nn, we must show there exists at least k other minimum bounding rectangles such that the $MinMaxDistance$ of each of them is less than the $MinDistance$ of M_i. These ideas are given below in Definition 2.

Definition 2 (Strategy 1 for k-nn). *Given a query point q, a list of minimum bounding rectangles $\mathcal{M}$, discard any minimum bounding rectangle $M_i \in \mathcal{M}$ if there exists $\mathcal{M}' \subset \mathcal{M}$ such that $|\mathcal{M}'| \geq k$ and for every $M_j \in \mathcal{M}'$ we have $MinDistance(q, M_i) > MinMaxDistance(q, M_j)$*

Note if k is greater than the branching factor of the R-tree, this strategy is not applicable because it will never prune a single node. In the next section we give the explicit generalization of pruning strategy 3 to k-nearest neighbors. We follow it with a proof that strategy 3 dominates strategy 1 even in the case of k-nearest neighbors.

3.2 Pruning Strategy 3

Strategy 3 compares the current neighbor estimate distance to the $MinDistance$ of each minimum bounding rectangle in the *activeBranchList*. Note that for 1-nearest neighbor searches, the neighbor estimate is an upper bound on distance from the query point. When a node is bound by a rectangle with $MinDistance$ greater than the current estimate, we know the node does not contain a spatial object closer to the query point. So, pruning the node is the correct strategy. The formal definition captures these ideas.

Definition 3 (Strategy 3 for 1-nn). *Given a query point q, a list of minimum bounding rectangles $\mathcal{M}$, and a neighbor estimate e, discard any minimum bounding rectangle $M_i \in \mathcal{M}$ if $MinDistance(q, M_i) > e$*

At this point, a number of clarifications are necessary. First, like (Hjaltason & Samet, 1999), we use a priority queue to store the current k-best nearest neighbor estimates. We refer to the closest k-best estimate as e_0 and the farthest as e_{k-1}. Since we now keep a priority queue of the k-best neighbors, we can only prune an MBR (and corresponding node) if its $MinDistance$ is greater than the farthest neighbor. Only one slight change to this definition is required when extending it to k-nearest neighbors.

Definition 4 (Strategy 3 for k-nn). *Given a query point q, a list of minimum bounding rectangles $\mathcal{M}$, and a priority queue of nearest neighbor estimates e, discard any minimum bounding rectangle $M_i \in \mathcal{M}$ if $MinDistance(q, M_i) > e_{k-1}$*

Both Cheung and Fu (1998) and Hjaltason and Samet (1999) provide proofs that pruning strategy 3 dominates pruning strategy 1 for the 1-nearest neighbor query. Since we are the first to give explicit representations of these strategies for k-nearest neighbors, we provide a proof for the more general case.

Theorem 1. *Generalized strategy 3 dominates generalized strategy 1 for k-nearest neighbor queries.*

Proof. Suppose we're performing a k-nearest neighbor search with query point q and strategy 1 prunes MBR M from the active branch list $\mathcal{ABL}$ of MBRs. From Definition 2 there exists $\mathcal{ABL}' \subset \mathcal{ABL}$ such that $|\mathcal{ABL}'| \geq k$ and every M' in $\mathcal{ABL}'$ has $MinMaxDistance(q, M') < MinDistance(q, M)$. Since for any MBR N, $MinDistance(q, N) \leq MinMaxDistance(q, N)$, each M' in $\mathcal{ABL}'$ will be searched before M because $\mathcal{ABL}$ is sorted by $MinDistance$. Because each M' in $\mathcal{ABL}'$ is guaranteed to contain a spatial object with actual distance less than an object found in M and since we have $|\mathcal{ABL}'| \geq k$ we know the farthest k-best estimate e_{k-1} has distance less than M. Therefore, from Definition 4, M would also be pruned using strategy 3. □

3.3 Pruning Strategy 2

Strategy 2 doesn't perform direct pruning, but instead, updates the nearest neighbor estimate when distance guarantees can be made. Put simply, if the $MinMaxDistance$ of a node is less than the current distance estimate, then we can update the estimate because the future descent into that node is guaranteed to contain a spatial object with actual distance from the query point less than the $MinMaxDistance$. We call the estimates in updates of this form *promises*[1] because the estimate is not an actual distance, but an upper bound on the distance. Moreover the estimate is a promise of, or place holder for, a spatial object that is as least as good the promise's prediction. This strategy is stated formally in Definition 5

Definition 5 (1-NN Strategy 2). *Given a query point q, a list of MBRs $\mathcal{M}$, and a neighbor estimate e, replace e with $MinMaxDistance(q, M_i)$ if there exist $M_i \in \mathcal{M}$ such that $MinMaxDistance(q, M_i) < e$.*

On the surface, transforming Definition 5 from 1-nearest neighbor to k-nearest neighbors seems straightforward. When we find a node whose $MinMaxDistance$ is less than the farthest k-best estimate, we should remove the farthest node from our priority queue and update it with the $MinMaxDistance$ estimate. However, this type of unrestrained updating creates a serious problem. Because the $k - nn$ search is depth-first, updating the neighbor estimates queue may result in distances referring to the same spatial object.

This has two direct manifestations. The first involves two distance estimates referring to the same spatial object but that object is yet to be found. The second, similar case occurs when a distance estimate refers to an actual spatial object in our neighbor estimates. We identify the former as the *multiple reference problem* because two promises refer to the same *promised* spatial object. We call the latter the *usefulness problem* since the promises have outlived their usefulness in the search process.

In order to understand these problems further, we provide a number of examples. Suppose we are performing a 3-nearest neighbor search and our current neighbor estimate distances are $[10, 20, 30]$. Now suppose we discover a node n with $MinMaxDistance = 17$. Using the strategy outlined above, we'd remove the farthest distance 30 from our k-best neighbor estimates and insert the promise with distance 17. This results in the k-best neighbor estimates $[10, 17, 20]$. Now suppose we descend into n and arrive at node n' having a $MinMaxDistance$ of 16. Again, following our previous strategy we remove the farthest estimate 20 and insert the promise of distance 16. So our k-best neighbors are now $[10, 16, 17]$. The problem here is that the $MinMaxDistance$ estimate 16 may refer to the same spatial object as the $MinMaxDistance$ estimate 17. That is, we have a case of the *multiple reference problem*.

[1] Our use of the term promise is similar to the definition given in (Armstrong et al., 1996).

The second manifestation is very similar to the first except the k-best neighbor estimates are updated with the distance an actual spatial object instead of an estimate. For example, suppose we are performing the same 3-nearest neighbor search with current neighbor estimate distances $[10, 20, 30]$. Again, suppose we discover a node n with $MinMaxDistance = 17$ and insert the promise into our estimates resulting in $[10, 17, 20]$. Later, descending into n, we find a spatial object with an actual distance of 15. Inserting this into our estimates results in $[10, 15, 17]$. The contention now is whether the spatial object with actual distance 15 is the same one referred to by the promise 17. If it isn't, there's nothing to worry about, but if it is, then the usefulness of the promise in our k-best estimates is past due. This is a case of the *usefulness problem*. In both examples our pruning strategy will return erroneous nearest neighbor information.

Both the multiple reference and usefulness problems do not occur in the 1-nearest neighbor case since there is always only one object under consideration, and moreover because we only maintain one estimate of that object's upper bound.

3.4 Promise-pruning

To address the problems in Section 3.3 we need to create an algorithm that generalizes the updating ideas of strategy 2, but avoids the *multiple reference* and *usefulness* problems. Said differently, we need an algorithm that performs with respect to two invariants.

1. No two promises refer to the same spatial object.
2. No promise exists after its usefulness.

It's clear the reference problem occurs because we descend into a node after placing a promise for that node in the k-best estimates. Similarly, the usefulness problem occurs because of the same scenario. So, one way to avoid both problems and honor the invariants, is to descend into a node only after its promises have been removed from the k-best estimates. In some cases, the promise is removed naturally because better, closer objects are discovered. In other cases, we need to remove the promise ourselves before descending into the node and searching its children. Applying these ideas to Definition 5 results in the following generalized strategies.

Definition 6 (k-NN Strategy 2a). *Given a query point q, a list of MBRs $\mathcal{M}$, and priority queue of neighbor estimates e, remove e_{k-1} and insert a promise with distance equal to $MinMaxDistance(q, M_i)$ if there exist $M_i \in \mathcal{M}$ such that $MinMaxDistance(q, M_i) < e_{k-1}$.*

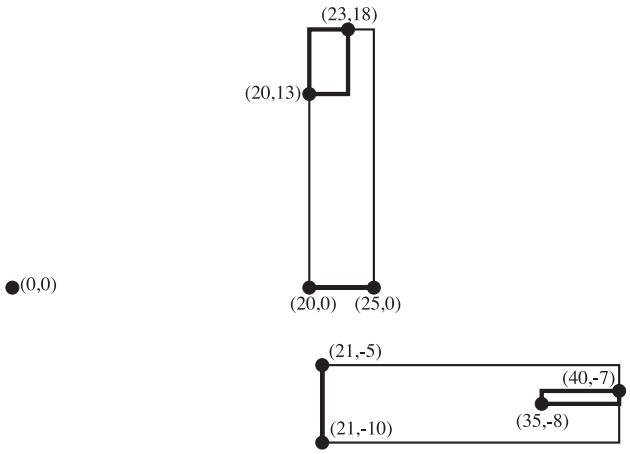


Figure 4: Eight 2-dimensional points, grouped with corresponding minimum bounding rectangles along with a query point at $(0, 0)$. Note that darker edges refer to tighter bounding MBRs.

Definition 7 (k-NN Strategy 2b). *Given a minimum bounding rectangle M and a priority queue of neighbor estimates e where applying Definition 6 to M resulted in a promise p placed in e, remove p from e (if possible) just prior to searching M.*

We've shown a technique for incorporating strategy 2 (and $MinMaxDistance$ information) into the pruning process, however it still may be unclear why simply removing a node's promise before searching it serves its maximal purpose. Recall that the goal of strategy 2 is to use $MinMaxDistance$ information to indirectly prune the search. That is, when distance information is known about future spatial objects, especially those that aren't directly in the search path, it should be incorporated into pruning. Knowing distance information about a node immediately prior to searching it is less informative because its children provide tighter bound estimates of the spatial objects. So, in some ways, one can view the forced removal of a promise as simply an opportunity to provide updated distance information.

The combination of Definitions 6 and 7 is a generalized algorithm for pruning strategy 2 we call *promise-pruning*.

4 Promise Pruning in k-Nearest Neighbor Queries

In the last section we described the transformation of pruning strategy 2 into an algorithm for promise pruning. In this section, we present the state-of-the-art depth-first k-nearest neighbor algorithm and trace its execution on a sample data set. Next we present the same algorithm augmented with promise pruning and show it prunes nodes that the unaugmented algorithm descended. That is, we provide a counterexample to past claims that pruning strategy 2 is not useful for depth-first k-nearest neighbor queries.

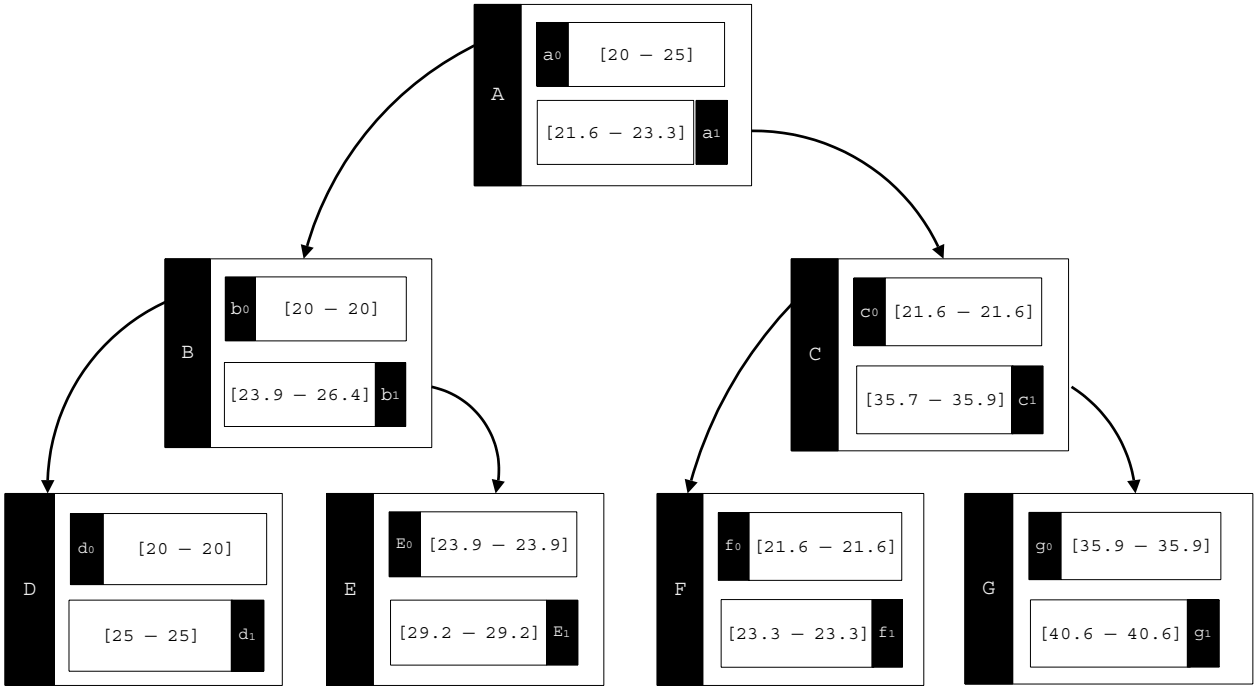


Figure 5: A possible R-tree for the points given in Figure 4. Each exterior rectangle denotes a node, while each interior rectangle denotes a record. Each record contains a pair of values $[a - b]$ with a representing the $MinDistance$ and b representing the $MinMaxDistance$ from the query point respectively.

4.1 Depth-First k-Nearest Neighbor Search Algorithm

Hjaltason and Samet (1999) provide the state-of-the-art depth-first k-nearest neighbor search algorithm given in Figure 6. Pruning strategy 3 is used exclusively for the search process. K-NN initializes *neighborEstimates* with infinity values and calls K-NNSEARCH with the root node of the R-tree. K-NNSEARCH checks if the current node is either internal or a leaf. When it's a leaf, the spatial objects' distances are examined for each record. An object is inserted into the *neighborEstimates* when it is closer to the query point than the current farthest object.

Suppose we have the R-tree given in Figure 5 formed from the data given in Figure 4. Note that each exterior rectangle of the R-tree denotes a node, while each interior rectangle denotes a record. The pair of numbers inside each record denotes the respective $MinDistance$ and $MinMaxDistance$ with respect to the query point $(0, 0)$.

Executing K-NN on the R-tree with query-point $(0, 0)$ results in the trace given in Figure 7. Note that K-NN visits all nodes in the R-tree except for G.

```
K-NN(query, rtree, k)
1.  FOR i from 0 to k-1
2.      estimate.dist ← ∞
3.      estimate.obj ← nil
4.      neighborEstimates_i ← estimate
5.  K-NNSEARCH(query, rtree.root, neighborEstimates)
6.  RETURN(neighborEstimates)

K-NNSEARCH(query, node, neighborEstimates)
1.  IF node.type = leaf
2.      FOREACH record in node.records
3.          WHEN DISTANCE(query, record.object) < neighborEstimates_{k-1}.dist
4.              REMOVELARGEST(neighborEstimates)
5.              estimate.dist ← DISTANCE(query, record.obj)
6.              estimate.obj ← record.obj
7.              INSERT(neighborEstimates,estimate)
8.  ELSE
9.      activeBranchList ← node.records
10.     sort activeBranchList by MinDistance
11.     FOREACH record in activeBranchList
12.         WHEN MINDISTANCE(query, record.mbr) < neighborEstimates_{k-1}.dist
13.             K-NNSEARCH(query, record.child, neighborEstimates)
```

Figure 6: The k-nearest neighbor algorithm given in (Hjaltason & Samet, 1999)

4.2 Depth-First k-Nearest Neighbor Search with Promise-Pruning

The algorithm K-NNSEARCH* in Figure 8 is identical to Figure 6 except it has been augmented with promise-pruning. It should be clear from the definition that K-NN* performs no worse than K-NN because the addition of promise-pruning can only decrease the number of nodes visited. Another view is that K-NN* is the generalization of the 1-nearest neighbor algorithm in Figure 1, to k-nearest neighbors with pruning strategies 1 and 3 combined.

For explanatory purposes, the algorithms in Figure 8 reflect promise-pruning without any performance enhancements. Although K-NNSEARCH* is asymptotically equivalent in running time to K-NNSEARCH, a number of small changes can help reduce overhead. For example, only the tail of *activeBranchList* needs to be promise-pruned since, if the first element of *activeBranchList* contains a potential promise, it will be removed before any descent occurs; hence the addition and removal are unnecessary steps. Finally, because we reinsert ∞ into the priority-queue when a promise is removed, using infinity for an underfull queue representation, while conceptually simple, can lead to unneeded operations. For example, if the priority queue is implemented as a heap, then inserting ∞ results in a worse case *heapify* operation. Simulating ∞

trace number	node	neighbor estimates	K-NNSEARCH steps
0	A	[∞, ∞]	8,9,10
1	a_0	[∞, ∞]	11,12,13
2	B	[∞, ∞]	8,9,10
3	b_0	[∞, ∞]	11,12,13
4	D	[∞, ∞]	1
5	d_0	[20, ∞]	2,3,4,5,6,7
6	d_1	[20, 25]	2,3,4,5,6,7
7	b_1	[20, 25]	11,12,13
8	E	[20, 25]	1
9	e_0	[20, 23.9]	2,3,4,5,6,7
10	e_1	[20, 23.9]	2
11	a_1	[20, 23.9]	11,12,13
12	C	[20, 25]	8,9,10
13	c_0	[20, 25]	11,12,13
14	F	[20, 25]	1
15	f_0	[20, 21.6]	2,3,4,5,6,7
16	f_1	[20, 21.6]	2
17	c_1	[20, 21.6]	11

Figure 7: A trace of the k-nn algorithm given in (Hjaltason & Samet, 1999) on the R-tree given in Figure 5 with $k = 2$

is easily accomplished by storing the priority-queue size and inserting items when appropriate.

Executing our K-NN* algorithm on the R-tree given in Figure 5 results in the trace shown in Figure 9. Notice that nodes G and E are never visited. This means *k-NN** pruned a node that *k-NN* did not. This example alone shows strategy 2, when generalized, is applicable to k-nearest neighbor queries and can potentially result in fewer node accesses.

In the next section we'll show empirically that the hand-crafted counterexample is not pathological. In fact, multiple experiments reveal promise-pruning reduces node accesses in both random and real-world data.

5 Experiments

In this section we verify the theoretical results from Section 4 with a number of empirical evaluations. We present three experiments; two with random data, and one with real data. Each experiment uses the R-tree implementation given in (Guttman, 1984) with the quadratic insertion algorithm.

```
K-NN*(query, rtree, k)
1.  FOR i from 0 to k-1
2.      estimate.dist ← ∞
3.      estimate.obj ← nil
4.      neighborEstimates_i ← estimate
5.  K-NNSEARCH*(query, rtree.root, neighborEstimates)
6.  RETURN(neighborEstimates)

K-NNSEARCH*(query, node, neighborEstimates)
1.  IF node.type = leaf
2.      FOREACH record in node.records
3.          WHEN DISTANCE(query, record.object) < neighborEstimates_{k-1}.dist
4.              REMOVELARGEST(neighborEstimates)
5.              estimate.dist ← DISTANCE(query, record.obj)
6.              estimate.obj ← record.obj
7.              INSERT(neighborEstimates,estimate)
8.  ELSE
9.      activeBranchList ← node.records
10.     sort activeBranchList by MinDistance
11.     PROMISEPRUNE(activeBranchList, neighborEstimates, query)
12.     FOREACH record in activeBranchList
13.         WHEN record.hasPromise AND record.promise is still in neighborEstimates
14.             REMOVE(neighborEstimates, record.promsie)
15.             INSERT(neighborEstimates,∞)
16.         WHEN MINDISTANCE(query, record.mbr) < neighborEstimates_{k-1}.dist
17.             K-NNSEARCH*(query, record.child, neighborEstimates)

PROMISEPRUNE(activeBranchList, neighborEstimates, query)
1.  FOREACH record in activeBranchList
2.      WHEN MINMAXDISTANCE(query, record.mbr) < neighborEstimates_{k-1}.dist
3.          REMOVELARGEST(neighborEstimates)
4.          promise.dist ← MINMAXDISTANCE(query, record.mbr)
5.          INSERT(neighborEstimates,promise)
6.          record.hasPromise ← TRUE
7.          record.promise ← promise
```

Figure 8: The k-nearest neighbor algorithm augmented with promise pruning.

trace number	node	neighbor estimates	K-NNSEARCH steps	PROMISEPRUNE steps
0	A	[∞, ∞]	8,9,10	
1	a_0	[25, ∞]	11	1,2,3,4,5,6,7
2	a_1	[23.3, 25]	11	1,2,3,4,5,6,7
3	a_0	[23.3, ∞]	12,13,14,15,16,17	
4	B	[23.3 ∞]	8,9,10	
5	b_0	[20, 23.3]	11	1,2,3,4,5,6,7
6	b_1	[20, 23.3]	11	1,2
7	b_0	[23.3, ∞]	12,13,14,15,16,17	
8	D	[23.3, ∞]	1	
9	d_0	[20, 23.3]	2,3,4,5,6,7	
10	d_1	[20, 23.3]	2,3	
11	b_1	[20, 23.3]	12,13,16	
12	a_0	[20, ∞]	12,13,14,15,16,17	
13	C	[20,∞]	8,9,10	
14	c_0	[20, 21.6]	11	1,2,3,4,5,6,7
15	c_1	[20, 21.6]	11	1,2
16	c_0	[20, ∞]	12,13,14,15,16,17	
17	F	[20, ∞]	1	
18	f_0	[20, 21.6]	2,3,4,5,6,7	
19	f_1	[20, 21.6]	2,3	
20	c_1	[20, 21.6]	12,13,16	

Figure 9: a trace of the k-nn algorithm augmented with promise pruning on the R-tree provided in Figure 5 with $k = 2$.

5.1 Experiment 1

In experiment 1 we inserted 10,000 different 2-dimensional points into an R-tree with maximum branching factor 10 and minimum branching factor 5. Each point occupied a unique location on the 2-dimensional integer grid from 1 to 100. That is, each point had the form (i, j) where i and j are integers in the closed interval $[1, 100]$. We performed 10000 k-nearest neighbor searches using each point in the tree as the query object. With $k = 31$, 6003 of the queries resulted in at least one saved node access. The overall distribution is provided below:

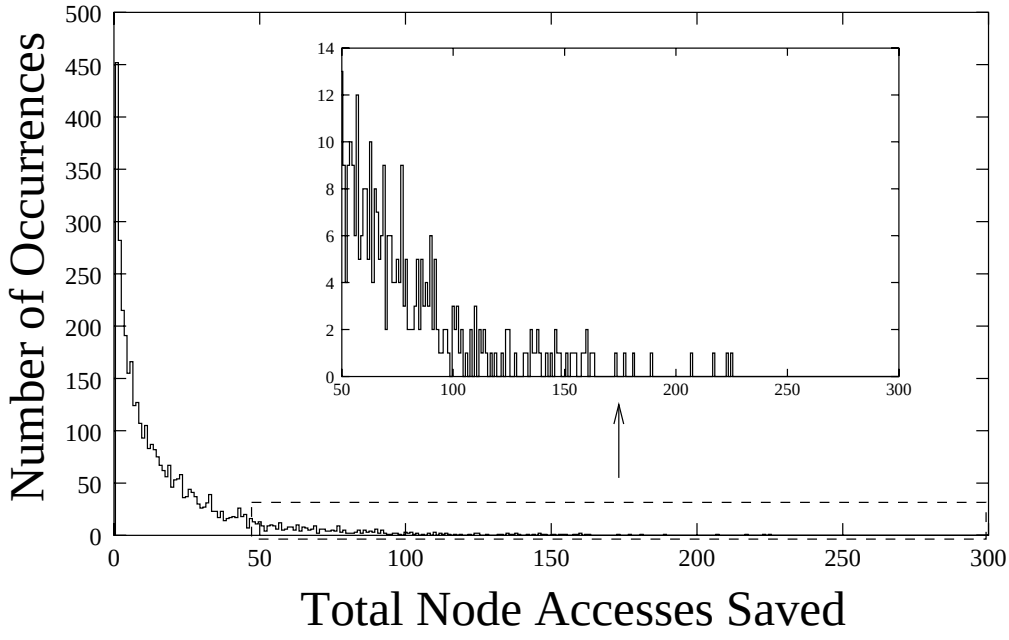
Total Node Accesses Saved	1	2	3	4	5	6	7	8
Number of Occurrences	3530	1269	662	261	183	64	18	16

Of those 6003 queries where promise-pruning helped, a majority saved only a total of one node access. However, the uniform nature of points and hence the queries, facilitates better, tighter groupings at the leaf node level. Still, over 60% of the queries decreased their node access count and if node accesses are considered expensive, then promise-pruning offers some savings.

5.2 Experiment 2

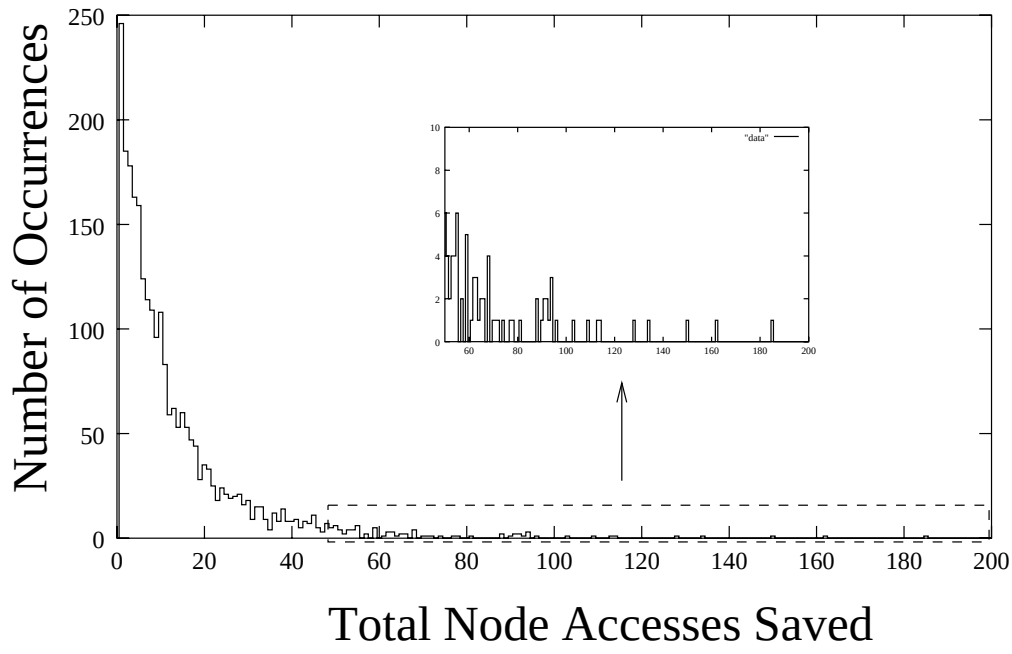
For experiment 2 we generated 50,000 10-dimensional points. Every coordinate of each point was constructed by independently sampling uniformly from the real interval $(-1000, 1000)$. We inserted these points into an R-tree with maximum branching factor 5 and minimum branching factor 2. Given this R-tree we performed two subexperiments.

In the first we performed 101 k-nearest neighbor queries (with $1 \leq k \leq 101$) on 100 10-dimensional points. Each point was located along the diagonal of the hyperplane starting with each axis fixed at 1 and ending with 100. The total number of queries then was 10100. 37% of the queries result in less node accesses when using promise-pruning. The distribution of total savings is given below.



The highest number of node accesses saved for a single query was 225. This occurred with a 42-nearest neighbor query on the point occupying the diagonal axises fixed at 90. Of the node saving queries, 452 saved a total of 1 node; 282 a total of 2 nodes; 215, a total of 3 nodes; and in general continued with, what appears to be, exponential decay.

In the second experiment we performed 99 k-nearest neighbor queries (with $2 \leq k \leq 100$) on 64 uniformly random 10-dimensional points. This amounts to a total of 6336 queries. 38% of the queries result in less node accesses when using promise-pruning. The distribution of total node accesses saved is given below.



The highest number of node accesses saved for a single query was 185. This occurred with a 74-nearest neighbor query on the random point $(261.0, 430.4, -620.4, -44.1, 84.3, -193.4, -826.3, 458.9, 202.3, -830.6)$. Of the node saving queries, 246 saved at total of 1 node; 185 a total of 2 nodes; and 178, a total of 3 nodes. In general, like the first experiment, a linear increase in node access savings corresponded with an apparent exponentially decaying number of queries.

5.3 Experiment 3

The third experiment uses data collected from 212 images captured by a sony pan-tilt-zoom camera mounted on a Pioneer 2 mobile robot. In each image, an object is tracked based on color, and the perimeter of that object in 2-dimensional space is recorded. The curvature scale space of each image is computed and the maxima locations of the curves are collected and inserted into a 15 dimensional point. In total then, we have 212 points.

We inserted the 212 15-dimensional points into an R-tree with maximum branching factor 5 and minimum branching factor 2. The query points for this experiment were the original 212 points themselves. For each point we performed 100 k-nearest neighbor queries (with k ranging from 1 to 100) both with and without promise-pruning. This means we performed a total of 42400 queries, of which half, 21200, were unique. In 721 cases, using promise-pruning resulted in node access savings. The table below gives the overall distribution:

Total Node Accesses Saved	1	2	3	4	5	7	15	16
Number of Occurrences	540	59	79	39	1	1	1	1

Only 3% of the queries resulted in a savings, however, this is almost certainly a result of having only 212 points in the R-tree. Still, even with sparse data, we see promise-pruning can have a positive effect.

6 Conclusion

We addressed some misconceptions in the literature about pruning strategies in nearest neighbor queries. We generalized three common 1-nearest neighbor pruning strategies to k-nearest neighbors and demonstrated that the transformation of strategy 2 is non-trivial. Two invariants were introduced that facilitated the discovery of a novel algorithm called *promise-pruning*. This procedure uses upper bound information to help guide and inform the search process. We showed that applying promise-pruning to the state-of-the-art depth-first k-nearest neighbor algorithm results in an overall, provably better search. These theoretical findings were reinforced with several empirical results that clearly showed promise-pruning aids the pruning process in R-trees.

7 Acknowledgements

Acknowledgements withheld for blind reviewing.

References

Armstrong, J., Virding, R., Wikstrom, C., & Williams, M. (1996). *Concurrent programming in Erlang*. Prentice Hall PTR.

Beckmann, N., Kriegel, H., Schneider, R., & Seeger, B. (1990). R*-tree: An efficient and robust access method for points and rectangles. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 322–331).

Cheung, K. L., & Fu, A. W.-C. (1998). Enhanced nearest neighbour search on the r-tree. *SIGMOD Record, 27*, 16–21.

Guttman, A. (1984). R-trees: A dynamic index structure for spatial searching. *Proceedings of the ACM SIGMOD International Conference on Management of Data* (pp. 47–57).

Hjaltason, G. R., & Samet, H. (1999). Distance browsing in spatial databases. *ACM Transactions on Database Systems, 24*, 265–318.

Roussopoulos, N., Kelley, S., & Vincent, F. (1995). Nearest neighbor queries. *Proceedings ACM SIGMOD Internaiontal Conference on the Management of Data* (pp. 71–79).

Sellis, T., Roussopoulos, N., & Faloutsos, C. (1988). R+-tree: A dynamic index for multidimensional objects. *Proceedings of the 13th International Conference on Very Large Databases* (pp. 507–518).