# Extending Range Queries and Nearest Neighbors

Robin Y. Flatland and Charles V. Stewart
Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY, 12180

## Abstract

*Given an initial rectangular range or k nearest neighbor (k-nn) query (using the $L_\infty$ metric), we consider the problems of incrementally extending the query by increasing the size of the range, or by increasing k, and reporting the new points incorporated by each extension. Although both problems may be solved trivially by repeatedly applying a traditional range query or $L_\infty$ k-nn algorithm, such solutions do not minimize the overall time to process all extensions. Our algorithms, however, obtain efficient overall query times by performing novel searches of multiple range trees and our related extending k-nn trees. In two dimensions, when queries eventually incorporate $\Theta(N)$ points or require $E = \Omega(N)$ extensions, the overall retrieval time of our algorithms is $O(E + N)$, which is optimal. Our extending $L_\infty$ k-nn algorithm immediately provides a new solution to the traditional $L_\infty$ k-nn problem, improving upon previous results.*

## 1 Introduction

Extending neighborhood problems, a class of problems generalizing the well-known range queries and $k$-nn problems, take a set of points in $R^d$ and ask for the new points incorporated by incrementally enlarging neighborhoods. In this paper we give efficient algorithms for two instances of extending neighborhood problems stated formally as follows:

**Extending Orthogonal Range Queries Problem:** Given a set of $N$ points in $R^d$ and an on-line sequence of $d$-dimensional, axis aligned, rectangular query regions $Q_1, \ldots, Q_E$, with each $Q_i$ completely containing $Q_{i-1}$, for the $i$(th) extended query, report the points in $Q_i$ that are not in $Q_{i-1}$.

**Extending $L_\infty$ $k$ Nearest Neighbors Problem:** Given a set of $N$ points in $R^d$, a query point $q$, and an on-line sequence of integers $k_1, \ldots, k_E$, with $0 < k_{i-1} < k_i \leq N$, for the $i$(th) extended query, report the $k_{i-1} + 1$(st) through $k_i$(th) nearest neighbors to $q$ using the $L_\infty$ [1] ($L_1$) metric.

Our focus is on minimizing the total time to process all $E$ extensions. We believe we are the first to explicitly consider these problems. Because we expect many sequences of extending queries on a static point set, our algorithms include a preprocessing stage to organize the points into a search structure which facilitates the processing of extending queries. Thus, we analyze each algorithm based on preprocessing time, storage, the time to process a single extension, and the overall time to process all $E$ extensions. Although both problems could be solved trivially by repeatedly applying a traditional orthogonal range query [1, 7, 3] or $L_\infty$ k-nn algorithm, our algorithms have asymptotically better overall extension times. Our results are summarized in Table 1 and described below. (See [6] for more detail.)

When only a single extension is made, our extending $k$-nn algorithm is immediately a new $L_\infty$ $k$-nn algorithm (see Table 1). In 2D, a minor modification reduces the storage requirement to $O(N)$. Our algorithm improves upon the best known $L_\infty$ $k$-nn algorithms. It has the same preprocessing and query time as Eppstein and Erickson [5]'s $L_\infty$ $k$-nn algorithm but requires less space and is more general because $k$ need not be fixed. Further, it solves the all-$k$-nearest-neighbors problem more efficiently than Dickerson *et al.* [4]'s $O(N \log N + kN \log k)$ algorithm, although ours is restricted to the $L_\infty$ ($L_1$) metric whereas theirs is for any convex distance function.

Extending neighborhood problems arise in computer vision surface reconstruction techniques that incrementally grow surfaces in 3D scene data as well as $k$-nn classification schemes that examine the neighbors in increasing order or that increase $k$ online. Because extending neighborhood problems are natural generalizations of the widely applicable range queries and $k$-nn problems, we suspect they arise in other applications as well.

## 2 Range Tree Background

Since our algorithms perform novel searches of range trees and our related extending $k$-nn trees, we begin by reviewing the range tree. A 2D range tree [1] for a planar point set $D$ is a balanced binary search tree ordered by $x$ coordinate with the points stored at the leaves. Each node $v$ stores the $x$ range of the points in its subtree's leaves and an array $Y(v)$ of the points $(x_i, y_i)$ at the leaves of its subtree ordered by $y$ coordinate. Each $(x_i, y_i)$ in $Y(v)$ has two *bridge* pointers [7] which connect it to the point in $Y(v \rightarrow lt)$

[1] Distances between two $d$ dimensional points $p$ and $q$ in the $L_\infty$ metric are given by $d(p,q) = max(|p_1-q_1|, |p_2-q_2|, ..., |p_d-q_d|)$. Since there is a linear time isometry from the $L_1$ to the $L_\infty$ metric, our algorithms also apply to the $L_1$ metric.

| Problem | Preprocessing | Storage | $i$(th) Extension | Overall Time for $E$ Extensions |
|---|---|---|---|---|
| 2D Extending Orthogonal Range Queries | $O(N \log N)$ | $O(N \log^{\epsilon} N)$ | $O(\log N + w_i)$ | $O(E \log(N/E) + E + w)$, for $E \leq N$ <br> $O(E + N)$, for all $E$ <br> optimal when $E = \Omega(N)$ or $w = \Theta(N)$ |
| 3D Extending Orthogonal Range Queries | $O(N \log^2 N)$ | $O(N \log^{1+\epsilon} N)$ | $O(\log^2 N + w_i)$ | $O((E \log(N/E) + E) \log N + w)$, for $E \leq N$ <br> $O(E + N \log N)$, for all $E$ |
| 2D Extending $L_\infty$ $k$-nn | $O(N \log N)$ | $O(N \log^{\epsilon} N)$ | $O(\log N + k_i - k_{i-1})$ | $O(min(E \log(N/E) + E + k_E, N))$, $E \leq N$ <br> optimal when $k_E = \Theta(N)$ |
| 3D Extending $L_\infty$ $k$-nn | $O(N \log^2 N)$ | $O(N \log^{1+\epsilon} N)$ | $O(\log^2 N + k_i - k_{i-1})$ | $O(min((E \log(N/E) + E) \log N + k_E, N \log N))$, $E \leq N$ |
| 2D $L_\infty$ $k$-nn | $O(N \log N)$ | $O(N)$ | $O(\log N + k)$ query time | |
| 3D $L_\infty$ $k$-nn | $O(N \log^2 N)$ | $O(N \log^{1+\epsilon} N)$ | $O(\log^2 N + k)$ query time | |

Table 1: Summary of results. $w_i$ is the number of points reported in the $i$(th) extension, $w = \sum_{i=1}^{E} w_i$, and $\epsilon$ is any real greater than 0.

[2] and the point in $Y(v \rightarrow rt)$ with largest $y$ coordinate less than or equal to $y_i$.

Given a rectangular query region $\{[x_l...x_u], [y_l...y_u]\}$, a range tree may be used to report all points $(x_i, y_i) \in D$ such that $x_l \leq x_i \leq x_u$ and $y_l \leq y_i \leq y_u$. The tree is searched from the root for the two leaf nodes $P(x_l)$ and $S(x_u)$. $P(x_l)$ is the leaf node whose stored point is the ***predecessor*** of $x_l$ in $D$, *i.e.* the point in $D$ with largest $x$ coordinate less than $x_l$; $S(x_u)$ is the leaf node whose stored point is the ***successor*** of $x_u$ in $D$, *i.e.* the point in $D$ with smallest $x$ coordinate greater than $x_u$. The two search paths determine at most $2 \log N$ ***basic*** nodes whose subtrees' leaves contain exactly those points in the $x$ query range. Letting $C$ be the last common ancestor of $P(x_l)$ and $S(x_u)$, the basic nodes are the right children of nodes on the path from $C$ to $P(x_l)$ and left children of nodes on the path from $C$ to $S(x_u)$ that are not path nodes themselves. At each basic node $b$, those points also in the $y$ query range are reported by scanning $Y(b)$ from the right of $y_l$'s predecessor in $Y(b)$. [3] The predecessor of $y_l$ in $Y(b)$ can be located in constant time at each $b$ if an initial binary search for the predecessor of $y_l$ in $Y(root)$ is performed and bridge pointers are followed down the search paths (since for every node $v$, the two bridge pointers associated with the predecessor of $y_l$ in $Y(v)$ point to the predecessor of $y_l$ in $Y(v \rightarrow lt)$ and $Y(v \rightarrow rt)$).

The range tree with bridge pointers requires $O(N \log^{d-1} N)$ preprocessing, $O(N \log^{d-1} N)$ storage, and $O(\log^{d-1} N + w)$ query time, where $w$ is the number of points reported. Chazelle's [3] compressed range trees reduce the storage to $O(N \log^{d-2+\epsilon} N)$, for any real $\epsilon > 0$; if the compressed range tree is used only for counting the points in the query region, the storage is further reduced to $O(N \log^{d-2} N)$.

## 3 Extending Orthogonal Range Queries

Our 2D extending orthogonal range queries algorithm uses two range trees, $T_{xy}$ and $T_{yx}$, to report the new points incorporated by each larger rectangular region. Tiling the area covered by the extended query into four rectangular regions as shown in Figure 1a, each tree efficiently reports points from two of the tiles. Although four queries (one for each tile) to a single range tree could be used to report the new points, such a solution requires $O(E \log N + w)$ overall extension time, where $w$ is the total number of points reported. Using two range trees, we obtain an asymptotically faster overall extension time which is optimal when $E = \Omega(N)$ or $w = \Theta(N)$.

$T_{xy}$ and $T_{yx}$ are built during preprocessing. $T_{xy}$ is a range tree as described in Section 2. $T_{yx}$ interchanges the roles of $x$ and $y$, *i.e.* it is a binary search tree ordered by $y$ coordinate with an array, call it $X(v)$, associated with each node $v$ ordered by $x$ coordinate. For query $Q_{i+1} = \{[x_l^{i+1}...x_u^{i+1}], [y_l^{i+1}...y_u^{i+1}]\}$, $T_{xy}$ reports points in regions $r_1^{i+1} = \{[x_l^{i+1}...x_l^i), [y_l^{i+1}...y_u^{i+1}]\}$ and $r_2^{i+1} = \{(x_u^i...x_u^{i+1}], [y_l^{i+1}...y_u^{i+1}]\}$. $T_{yx}$ reports points in regions $r_3^{i+1} = \{[x_l^i...x_u^i], [y_l^{i+1}...y_l^i)\}$ and $r_4^{i+1} = \{[x_l^i...x_u^i], (y_u^i...y_u^{i+1}]\}$.

We describe our algorithm inductively for query $Q_{i+1}$ assuming query $Q_i$ has been processed and pointers are available to leaf nodes $P(x_l^i)$ and $S(x_u^i)$ in $T_{xy}$ and $P(y_l^i)$ and $S(y_u^i)$ in $T_{yx}$. ($P(y_l^i)$ and $S(y_u^i)$ are defined similarly as $P(x_l^i)$ and $S(x_u^i)$ but w.r.t. the $y$ coordinate.) Query $Q_1$, our base case, is handled uniquely. It is processed as a normal range query in both $T_{xy}$ and $T_{yx}$. Initial binary searches for $y_l^1$'s predecessor in $T_{xy}$'s $Y(root)$ array and for $x_l^1$'s predecessor in $T_{yx}$'s $X(root)$ array are performed and bridge pointers propagate this information down the tree. The four search paths terminate at $P(x_l^1)$ and $S(x_u^1)$ in $T_{xy}$ and $P(y_l^1)$ and $S(y_u^1)$ in $T_{yx}$.

For query $Q_{i+1}$, the points contained in region $r_1^{i+1}$ are reported by traversing the path up and back down $T_{xy}$ from leaf node $P(x_l^i)$ to leaf node $P(x_l^{i+1})$ using the $x$ range stored at each node to determine when to descend. The search path determines at most $2 \log N$ basic nodes whose subtrees' leaves contain exactly the points in the range $[x_l^{i+1}...x_l^i)$ of region $r_1^{i+1}$ (see Figure 1b). On the way up the tree these nodes are left children nodes, not on the path themselves, whose parents are on the path. On the way down these nodes are right children nodes, not on the path them-

[2] $v \rightarrow lt$ and $v \rightarrow rt$ indicate the left and right children of $v$.

[3] Here predecessor is defined the same as before but w.r.t. the $y$ coordinate and the set of points in $Y(b)$.

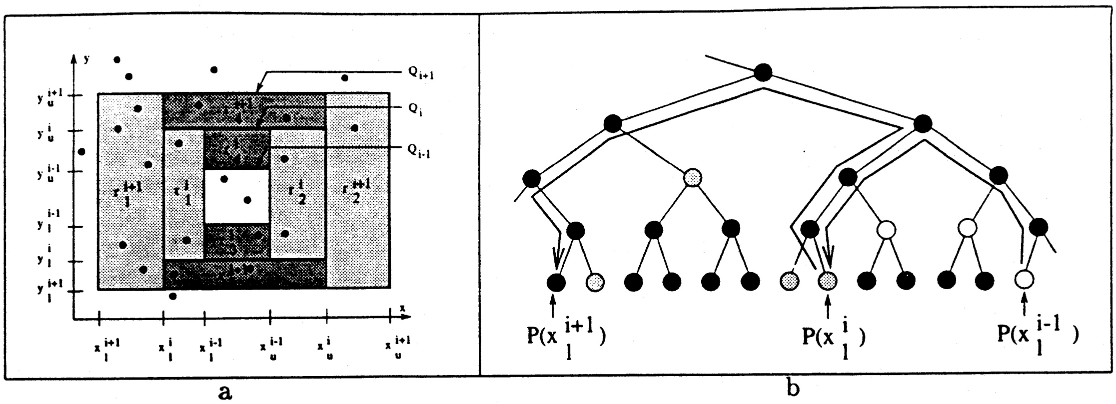


Figure 1: Two consecutive extended queries (a) and the corresponding searches of $T_{xy}$ (b) for reporting points in regions $r_1^i$ and $r_1^{i+1}$. The basic nodes for regions $r_1^i$ and $r_1^{i+1}$ are unfilled and light gray, respectively.

selves, whose parents are on the path. Leaf node $P(x_l^i)$ is also included if $P(x_l^{i+1}) \neq P(x_l^i)$.

The $Y$ array of each basic node $b$ is searched for points in the $y$ range $[y_l^{i+1}...y_u^{i+1}]$ of $r_1^{i+1}$ by locating the predecessor of $y_l^1$ in $Y(b)$ and scanning to the left and right from this item since $y_l^{i+1} \leq y_l^1 \leq y_u^{i+1}$. Locating the predecessor can be done in constant time if during all extensions including $Q_1$, whenever the search moves down the tree, the bridge pointers are followed and a pointer to the predecessor of $y_l^1$ in $Y(v)$ is stored at all visited nodes $v$. The stored predecessor pointers are used to find the predecessor at nodes when the search moves back up the tree. Using this, the total time to report the $w_1^{i+1}$ points in region $r_1^{i+1}$ is $O(\log N + w_1^{i+1})$.

Tree $T_{xy}$ is also used to report the points contained in region $r_2^{i+1}$ by traversing the path from leaf node $S(x_u^i)$ to $S(x_u^{i+1})$ and scanning the $Y$ arrays of the basic nodes to report the points in the $y$ range of $r_2^{i+1}$. Reporting points in regions $r_3^{i+1}$ and $r_4^{i+1}$ is analogous to regions $r_1^{i+1}$ and $r_2^{i+1}$, but using tree $T_{yx}$ instead of $T_{xy}$, interchanging the roles of $x$ and $y$. To keep the tiles from overlapping, points from the $X$ arrays in the $x$ range $[x_l^i...x_u^i]$ of query $Q_i$ are reported (see Figure 1a).

After completing the searches of $T_{xy}$ and $T_{yx}$, pointers to the four leaf nodes where extension $i$+2's searches begin are available since extension $i$+1's four searches terminate at these nodes.

**Theorem 3.1** *For the $i$(th) extension, the $w_i$ appropriate points are reported in $O(\log N + w_i)$ time. The overall time to process all $E$ extensions is $O(E\log(N/E)+E+w)$ when $E \leq N$ and $O(N+E)$ regardless of whether $E \leq N$ or $E > N$, where $w = \sum_{i=1}^{E} w_i$. This is optimal when $w = \Theta(N)$ or $E = \Omega(N)$.*

**Proof:** Clearly query $Q_1$ is processed in $O(\log N + w_1)$ time. All other extensions traverse four paths of length $O(\log N)$ up and back down the tree spending time at each node proportional to the number of points reported or constant time if no points are reported. Thus the worst case time for the $i$(th) extension is $O(\log N + w_i)$.

For the overall extension time when $E \leq N$, we consider the longest possible walk in $T_{xy}$ for reporting points in regions $r_1^i$, $i = 2...E$. The walk starts at $P(x_l^1)$ and moves up the tree and back down to $P(x_l^2)$, and from $P(x_l^2)$ to $P(x_l^3)$, and so on. Because the walk moves from right to left across the tree from one leaf node to another, at most one extension traverses a path of length $2\lceil\log N\rceil$ through the root, at most two extensions traverse paths of length $2(\lceil\log N\rceil - 1)$ through subtrees rooted at depth 1 in the tree, and in general, at most $2^i$ traverse paths of length $2(\lceil\log N\rceil - i)$ through subtrees rooted at depth $i$. For each path node, at most one basic node is visited. Assuming without loss of generality that $E$ is a power of 2, an upper bound on the total number of path nodes and basic nodes visited is

$$2\sum_{i=0}^{\log E-1} 2^i[2(\lceil\log N\rceil - i) + 1] \quad (1)$$

which is $O(E\log(N/E) + E)$. Reporting points from the other three regions yield the same worst case walks. The time spent at each node is proportional to the number of points reported or is constant if no points are reported. The overall extension time when $E \leq N$ is then $O(E\log(N/E) + E + w)$, noting that query $Q_1$ is a special case covered by this asymptotic bound.

The overall extension time regardless of whether $E \leq N$ or $E > N$ is bounded by $O(N + E)$. Each of the $O(N)$ internal nodes of $T_{xy}$ and $T_{yx}$ is visited at most a constant number of times since the walks in $T_{xy}$ and $T_{yx}$ are partial tree traversals moving from right to left (or left to right) across the trees. No more than two leaf nodes are visited per walk on any extension, so at most $O(E)$ leaf nodes are visited overall. Therefore, the overall extension time is at most $O(E + N)$. When $w = \Theta(N)$ or $E = \Omega(N)$, this is optimal since minimally we must report the points, and minimally we must spend a constant amount of time processing each extension. □

Preprocessing and storage are asymptotically bounded by the requirements of the two range trees which, using Chazelle's compressed range trees, are respectively $O(N \log N)$ and $O(N \log^{\epsilon} N)$, for any real $\epsilon > 0$. Generalizing this algorithm to three dimensions is straightforward using three 3D range tree data structures. Table 1 summarizes the results.

## 4 Extending $L_\infty$ $k$ Nearest Neighbors

In two dimensions, the $k_{i+1}$ $L_\infty$ nearest neighbor of query point $q$ defines a square region in the plane centered at $q$ containing all points at least as close to $q$ as itself. We call this the $k_{i+1}$ nearest neighbor square. For the $(i+1)(th)$ extension, the $k_i + 1$(st) to the $k_{i+1}$(th) nearest neighbors of $q$ are exactly the points contained in the $k_{i+1}$-nn square, not also in the $k_i$-nn square [4] (see Figure 2a). Using a novel interleaved search, our algorithm determines the $k_{i+1}$-nn square and makes an extending orthogonal range query to report the appropriate points.

During preprocessing two $2D$ *extending k-nn trees*, $T_x$ and $T_y$, are constructed. $T_x$ is a balanced binary search tree ordered by $x$ coordinate with the points stored at the leaves. Each node $v$ is augmented with two ordered arrays each containing the points stored at $v$'s subtree's leaf nodes. The first array, $A^-(v)$, contains the points in the order they are encountered by $L^-$, a 135° line swept across the plane from top to bottom. The second array, $A^+(v)$, contains the points in the order they are encountered by $L^+$, a 45° line swept across the plane from top to bottom. Bridge pointers are used to connect the items in $A^-(v)$ to the corresponding items in $A^-(v \rightarrow lt)$ and $A^-(v \rightarrow rt)$, and similarly for $A^+(v)$. $T_y$ is identical to $T_x$, but its search tree is ordered by $y$ coordinate. Due to the strong structural similarity between extending $k$-nn trees and range trees and due to the fact that we will use these trees for counting only, they can be constructed in $O(N \log N)$ time and stored in $O(N)$ space in the same way as Chazelle's [3] compressed range trees when used for counting only.

These trees are used to count points in wedge shaped regions of the plane. Query point $q$ partitions the plane into four quadrants ($L, R, T$, and $B$) defined by a 135° line, $L_q^-$, and a 45° line, $L_q^+$, passing through $q$ (see Figure 2a). Each node $v$ in $T_x$ defines a vertical slab in the plane which includes the region on and between vertical lines through the points stored at the leftmost and rightmost leaves of $v$'s subtree. If $v$ is a leaf node, then its slab is just a vertical line passing through its point. Within the slab lie exactly the points at the leaves of $v$'s subtree. For slabs (strictly) to the left of $q$, arrays $A^-(v)$ and $A^+(v)$ are used to count the points in the wedge formed by intersecting the slab with quadrant $L$ (see Figure 2b). This is done by locating the predecessor of $L_q^-$ in $A^-(v)$ (the point in $A^-(v)$ encountered by sweep line $L^-$ immediately before $L_q^-$) and the successor of $L_q^+$ in $A^+(v)$ (the point in $A^+(v)$ encountered by sweep line $L^+$ immediately after $L_q^+$). This gives us the number of slab points on or below $L_q^-$ and the number below $L_q^+$. The difference is the number of points in the wedge. Just as predecessor information was propagated around the range trees in Section 3, the predecessor of $L_q^-$ in $A^-(v)$ and successor of $L_q^+$ in $A^+(v)$ can be located in constant time at visited node $v$ if initial searches of the root arrays $A^-(root)$ and $A^+(root)$ are performed, bridge pointers are followed, and pointers to the successor and predecessor points are stored at visited nodes. Thus counting takes only constant time at each visited node. Similarly, constant time counting can be done for the slabs in the other quadrants.

During the search for the $k_{i+1}$-nn square, we maintain four pointers, $t_j$, $j \in \{L, R, T, B\}$; $t_L$ and $t_R$ point to nodes in $T_x$, and $t_T$ and $t_B$ point to nodes in $T_y$. The vertical slabs defined by $t_L$ and $t_R$ and the horizontal slabs defined by $t_T$ and $t_B$ will always lie (strictly) to the left, right, above, and below $q$, respectively. Associated with the node currently pointed to by $t_j$ are two regions and a count (see Figure 2c): $S_{t_j}$ is the square region whose boundary is all points equidistant from $q$ as the side of $t_j$'s slab farthest from $q$ intersected with quadrant $j$; $C_{t_j}$ is the number of the points in $S_{t_j}$; and $\Delta_{t_j}$ is the triangular region formed by intersecting $S_{t_j}$ with quadrant $j$. Finally, we maintain for each quadrant a count $c_j$ which is the number of points in $\Delta_{t_j}$. Each time $t_j$ moves in its tree, $c_j$ will be updated using our mechanism for counting points in wedges.

The goal in the $(i+1)$(th) extension is to "fix" the four quadrants with respect to the $k_{i+1}$-nn square. We describe what it means for quadrant $L$ to be fixed; the other three quadrants are analogous. Let $l_1, l_2, ..., l_N$ be the leaf nodes of $T_x$ ordered by increasing $x$ coordinate. Then quadrant $L$ is fixed when $t_L$ points to the leaf node $t_L = l_i$ (whose slab lies to the left of $q$) satisfying the inequality,

$$C_{l_i} \geq k_{i+1} > C_{l_{i+1}}.$$

When $L$ is fixed, the leaf node $t_L = l_i$ defines the square $S_{l_i}$ which is at least as large as the $k_{i+1}$-nn square and hence $C_{l_i} \geq k_{i+1}$. The leaf node $l_{i+1}$ immediately to its right defines a square smaller than the $k_{i+1}$-nn square and hence $C_{l_{i+1}} < k_{i+1}$. (Figure 2d shows the slab (a line) and corresponding square for every leaf node whose slab is located to the left of $q$. The square fixing quadrant $L$ is indicated.) When all four quadrants are fixed, the smallest $S_{t_j}$, $j \in \{L, R, T, B\}$, is the $k_{i+1}$-nn square. A quadrant which is not yet fixed is said to be *free*.

Our algorithm performs four searches of $T_x$ and $T_y$ for the four leaf nodes whose corresponding squares fix the four quadrants w.r.t. the $k_{i+1}$-nn square. The search begins with $t_j$, $j \in \{L, R, T, B\}$, pointing to the leaf node whose square fixes quadrant $j$ w.r.t. the $k_i$-nn square and the counts $c_j$ are known. Each step of quadrant $j$'s search is determined by evaluating the inequality $C_{t_j} \geq k_{i+1}$. (We discuss evaluating this equality below.) Specifically for quadrant $L$, if

[4] For clarity of explanation, we assume the points are in general position. Specifically, no two points are equidistant from $q$ and no two points share the same $x$ or $y$ coordinates. Removing this assumption requires only minor modifications to the algorithm.

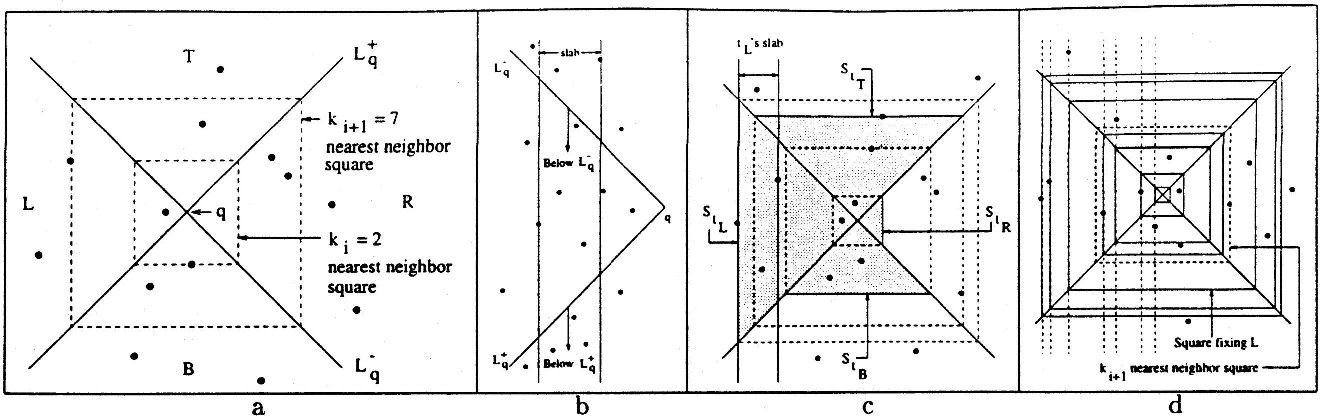


Figure 2: (a) Two nearest neighbor squares and the four quadrants defined by $q$. (b) Intersection of quadrant $L$ and a vertical slab. (c) The four squares defined by $t_j$, $j \in \{L, R, T, B\}$; only $t_L$'s slab is shown. $c_j$ is the number of points in the shaded triangular region in quadrant $j$. (d) Squares corresponding to each leaf node whose slab lies to the left of $q$. The square fixing quadrant $L$ w.r.t. the $k_{i+1}$-nn square is indicated.

$C_{t_L} < k_{i+1}$, $t_L$ moves up the tree towards the root to the first node whose left child is not on the upward path, and then down to this left child, making $t_L$ point to a node whose slab is adjacent to $t_L$'s old slab with $S_{t_L}$ larger than $t_L$'s old square. This is repeated until $C_{t_L} \geq k_{i+1}$ at which time the leaf node fixing $L$ lies either in $t_L$'s left or right subtree. Trying first to the right, $t_L$ moves to the right child whose slab is a subset of its parent's slab with $S_{t_L}$ smaller than its parent's square. If $C_{t_L} \geq k_{i+1}$, then the leaf node fixing $L$ must lie in $t_L$'s subtree, otherwise it lies in $t_L$'s left sibling's subtree. The search continues down the tree in this manner to the leaf node fixing $L$.

If the four searches were performed independently, evaluating the inequality $C_{t_j} \geq k_{i+1}$ at visited nodes would essentially require an $O(\log N)$ range query resulting in $O(\log^2 N)$ search time. But using our counts $c_j$, we can always evaluate the inequality in constant time for one of the free quadrants allowing that quadrant to advance its search one step. Our interleaved search *repeatedly determines the quadrant for which the inequality can be evaluated in constant time and advances that quadrant's search by one step until all four quadrants become fixed*, i.e. *all searches reach the leaf nodes fixing their quadrants*. Specifically, while all four quadrants are free, let $S_{t_c}$ be the current square whose boundary is closest to $q$ of the four squares $S_{t_j}$, $j \in \{L, R, T, B\}$, and let $S_{t_f}$ be the square whose boundary is farthest from $q$ (see Figure 2c where $c = R$ and $f = L$). Because $S_{t_c}$ is the smallest square, it is enclosed in the union of the current four triangular regions $\Delta_{t_j}$, $j \in \{L, R, T, B\}$, and so $\sum c_j \geq C_{t_c}$. Also, because $S_{t_f}$ is the largest square, it encloses this union, and so $\sum c_j \leq C_{t_f}$. Hence, if $\sum c_j < k_{i+1}$ then $C_{t_c} < k_{i+1}$, implying that square $S_{t_c}$ is smaller than the $k_{i+1}$-nn square and quadrant $c$ advances its search one step by moving $t_c$ up the tree to the next adjacent slab, thus enlarging $c$'s square. (For example, if $c = L$ then $t_L$ moves up the tree towards the root to the first node whose left child is not on the upward path, and then $t_L$ moves to this left child.) Alternatively, if $\sum c_j \geq k_{i+1}$ then $C_{t_f} \geq k_{i+1}$, implying that square $S_{t_f}$ is at least as big as the $k_{i+1}$-nn square. At this time, the leaf node fixing quadrant $f$ lies either in $t_f$'s left or right subtree. Trying the subtree with the smaller square first, quadrant $f$ advances its search one step by moving $t_f$ down one level in its tree, thus shrinking $f$'s square. (For example, if $c = L$ then $t_L$ moves down to the right child. Note that if the leaf node fixing quadrant $L$ actually lies in $t_L$'s left sibling's subtree, eventually $S_{t_L}$ will become the smallest free square with $C_{t_L} < k_{i+1}$ and it follows that $t_L$ will move up the tree to its left sibling at that time.)

Using our counting technique, $c_j$ can be updated in constant time each time the search is advanced in quadrant $j$. For example, when $t_L$ moves up the tree, $c_L$ is incremented by the number of points in the intersection of $t_L$'s new slab and quadrant $L$. When $t_L$ moves down the tree to its right child, $c_L$ is decremented by the number of points in the intersection of $t_L$'s sibling's slab and quadrant $L$.

When one or more of the quadrants become fixed (meaning the search in that quadrant has completed), this scheme requires a slight modification. Fortunately, we will still be able to evaluate the inequality $C_{t_j} \geq k_{i+1}$ for either the largest or the smallest square in a remaining free quadrant. To show this, we first make some observations. By definition, for any fixed quadrant $j$, $C_{t_j} \geq k_{i+1}$ and $C_{t'_j} < k_{i+1}$, where $t'_j$ is the leaf node immediately to the right of $t_j$ if $j \in \{L, B\}$ and the leaf node immediately to its left if $j \in \{R, T\}$. The square $S_{t_j}$ is at least as large as the $k_{i+1}$-nn square, and the square $S_{t'_j}$ is smaller than it. No points are located between the slab lines corresponding to nodes $t_j$ and $t'_j$. Further, **any** square $S_{t_k}$, $k \in \{L, R, T, B\}$, as small or smaller than $S_{t'_j}$ must have $C_{t_k} < k_{i+1}$, and **any** square $S_{t_k}$ as large or larger than $S_{t_j}$ must have $C_{t_k} \geq k_{i+1}$.

To determine which quadrant will execute the next step of its search when at least one quadrant is fixed, consider three cases. (1) There is a free quadrant $k$ and a fixed quadrant $j$, with square $S_{t_k}$ as large or larger than $S_{t_j}$. (2) There is a free quadrant $k$ and a fixed quadrant $j$, with square $S_{t_k}$ as small or smaller than square $S_{t'_j}$. (3) For all free quadrants $k$ and all fixed quadrants $j$, square $S_{t_k}$ is smaller than square $S_{t_j}$ and larger than square $S_{t'_j}$. (Cases (1) and (2) are not mutually exclusive.) In case (1), following from our observations above, we know that $C_{t_k} \geq C_{t_j} \geq k_{i+1}$, determining the next step in quadrant $k$. In case (2), again following from our observations above, we know that $C_{t_k} \leq C_{t'_j} < k_{i+1}$, determining the next step in quadrant $k$. For case (3), we calculate $r = \sum c_k + \sum c_{j'}$, where $k$ represents free quadrants, $j$ represents fixed quadrants, and $c_{j'}$ is the number of points in $t'_j$'s triangular region. Count $c_{j'}$ can be obtained from $c_j$ in constant time by subtracting one if $t_j$'s point lies in quadrant $j$, $c_j = c_{j'}$ otherwise. For any fixed quadrant $j$, observe that any square smaller than $S_{t_j}$ and larger than $S_{t'_j}$ will contain exactly $c_{j'}$ points in quadrant $j$. Thus, if $S_{t_f}$ is the largest free square and $S_{t_c}$ is the smallest free square, $C_{t_f} \geq r \geq C_{t_c}$. Hence if $r \geq k_{i+1}$, then $C_{t_f} \geq k_{i+1}$, determining the next step of the search in quadrant $f$. If, however, $r < k_{i+1}$, then $C_{t_c} < k_{i+1}$, determining the next step of the search in quadrant $c$. Thus, by comparing $r$ to $k_{i+1}$ the next step in one of the free quadrants may always be determined.

The interleaved search is complete when all four quadrants are fixed. The smallest square is the $k_{i+1}$-nn square and an extending range query reports the points. The data structures are now ready to find the $k_{i+2}$-nn square. Specifically, pointers $t_j$ point to the leaf nodes whose squares fix their quadrants w.r.t. the $k_{i+1}$-nn square and counts $c_j$ are known.

The only part of the algorithm remaining is initializing the data structures in preparation for the first extension. Given $q$, we initialize our extending $k$-nn search by fixing the four quadrants w.r.t. the first nearest neighbor square. To do this, the first nearest neighbor to $q$ in each quadrant is located in $O(\log N)$ time by searching the appropriate extending $k$-nn tree using counts $c_j$ to guide the search. The point closest to $q$ of these four points is $q$'s nearest neighbor and defines the nearest neighbor square. Each quadrant is then fixed with respect to this square in $O(\log N)$ time by searching $T_x$ or $T_y$ for the appropriate leaf node and making $t_j$ point to it. Counts $c_j$ are initialized to either zero or one depending on whether or not the point on $t_j$'s slab line lies in quadrant $j$. (See [6] for more detail.)

**Theorem 4.1** *For the i(th) extension, the $k_i - k_{i-1}$ appropriate points are reported in $O(\log N + k_i - k_{i-1})$ time. The overall time to process all $E$ extensions is $O(min(E\log(N/E) + E + k_E, N))$, which is optimal when $k_E = \Theta(N)$.*

**Proof:** Initializing the search by fixing the four quadrants w.r.t. the first nearest neighbor square takes $O(\log N)$ time. All other extensions traverse four paths of length $O(\log N)$ up and back down the tree spending constant time counting at each node. Thus the worst case time to find the $k_i$-nn square is $O(\log N)$. Reporting the points requires $O(\log N + k_i - k_{i-1})$ time by Theorem 3.1. Noting that $E$ cannot be larger than $N$ from the problem definition, a proof similar to that in Theorem 3.1 shows the upper bound on the total number of nodes visited is $O(E\log(N/E) + E)$, with constant time spent at each node. Reporting the points requires $O(E\log(N/E) + E + k_E)$ time by Theorem 3.1. The overall extension time is also bounded by $O(N)$ with proof similar to Theorem 3.1. When $k_E = \Theta(N)$, this is optimal since minimally we must report the points. □

Preprocessing and storage are dominated by the requirements of the extending range queries algorithm. Generalizing our extending $k$-nn algorithm to three dimensions is fairly straightforward using three 3D extending $k$-nn trees. Table 1 summarizes the results.

## 5 $L_\infty$ $k$-nn Algorithm

Our extending $L_\infty$ $k$-nn algorithm is immediately a new $L_\infty$ $k$-nn algorithm by setting $E = 1$ and $k_1 = k$. In two dimensions, we can reduce the storage requirements by using Chazelle's 2D, fixed aspect ratio, range reporting algorithm [2] to report the points in $O(\log N + k)$ time, requiring $O(N \log N)$ preprocessing and only $O(N)$ space. As stated in the introduction, this improves upon related but more restrictive 2D $L_\infty$ $k$-nn algorithms. Since we are not aware of any such fixed aspect ratio range reporting algorithm for 3D queries, our new 3D $k$-nn algorithm has the same preprocessing, storage, and query time as our 3D extending $k$-nn algorithm when $E = 1$ and $k_1 = k$.

## References


[1] J. L. Bentley. Multidimensional divide-and-conquer. *Communications of the ACM*, 23:214–229, 1980.

[2] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. In *Proceedings of the 2$^{nd}$ Annual ACM Symposium on Computational Geometry*, pages 293–302, 1986.

[3] Bernard Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal on Computing*, 17:3:427–462, 1988.

[4] Matthew T. Dickerson, R. L. Scot Drysdale, and Jorg-Rudiger Sack. Simple algorithms for enumerating interpoint distances and finding $k$ nearest neighbors. *International Journal of Computational Geometry and Applications*, 2:221–239, 1992.

[5] David Eppstein and Jeff Erickson. Iterated nearest neighbors and finding minimal polytypes. *Discrete and Computational Geometry*, 11:321–350, 1994.

[6] Robin Y. Flatland and Charles V. Stewart. Extending range queries and nearest neighbors. Technical report, Rensselaer Polytechnic Institute, 1994.

[7] Dan E. Willard. New data structures for orthogonal range queries. *SIAM Journal on Computing*, 14:232–253, 1985.