Range trees and the post-office problem

Michiel Smid[*]

May 20, 1999

This chapter discusses the following problem from computational geometry.

Problem 1 *Given a set S of n points in $\mathbb{R}^D$, preprocess them into a data structure such that for any query point $p \in \mathbb{R}^D$, we can efficiently find a point $p^* \in S$ that is nearest to p, i.e.,*

$$d(p, p^*) = \min\{d(p,q) : q \in S\}.$$

Here, $d(p,q)$ denotes the Euclidean distance between p and q:

$$d(p,q) = \left(\sum_{i=1}^{D} (p_i - q_i)^2 \right)^{1/2}.$$

This problem is known as the *nearest neighbor searching problem* or *post-office problem*: Think of S as a set of post-offices. Assume you are walking around. Suddenly, you find a letter in your pocket which you want to send. At that moment, you want to know the post-office that is closest to your current position.

In the planar case, i.e., $D = 2$, the problem can be solved optimally, i.e., with $O(\log n)$ search time and using $O(n)$ space, using Voronoi diagrams and point location. In higher dimensions, however, the problem gets difficult. The best known results either use a large amount of space (roughly $n^{D/2}$) or have a very high search time (roughly $n^{1-f(D)}$, where $f(D)$ goes to zero for increasing D).

In view of this negative result, it is natural to consider weaker versions of the post-office problem. What happens to the complexity of the problem if

[*]Fakultät für Informatik, Otto-von-Guericke-Universität Magdeburg. E-mail: `michiel@latrappe.cs.uni-magdeburg.de`.

we replace the Euclidean metric by a simpler metric? Define the L_∞-*distance* between the points p and q by

$$d_\infty(p,q) := \max\{|p_i - q_i| : 1 \leq i \leq D\}.$$

(The Euclidean distance is also called L_2-*distance.*) In the L_∞-*post-office problem*, we want to find a point $p^\infty \in S$ that is closest to the query point p w.r.t. the L_∞-metric, i.e.,

$$d_\infty(p, p^\infty) = \min\{d_\infty(p,q) : q \in S\}.$$

We can also consider the following *approximate L_2-post-office problem*: Let $\varepsilon > 0$ be a fixed constant. Instead of searching for the exact Euclidean neighbor p^* of p, we are satisfied with a $(1+\varepsilon)$-*approximate neighbor* of p, i.e., a point $q \in S$ such that

$$d_2(p,q) \leq (1+\varepsilon) \cdot d_2(p,p^*).$$

In this chapter, we will see that both these problems can be solved efficiently, i.e., with polylogarithmic search time, using $O(n(\log n)^{O(1)})$ space.

The data structure used is the *range tree*, one of the oldest data structures in computational geometry. Range trees were invented by Lueker (1978) and Bentley (1979) for solving the so-called *orthogonal range searching problem.* We show that they can also be used to solve the L_∞-post-office problem.

In the rest of this chapter, we restrict ourselves to the planar case. The generalization to higher dimensions is straightforward.

1 From the exact L_∞-problem to the approximate L_2-problem

Let S be a set of n points in the plane. In this section, we show that any solution for the L_∞-post-office problem can be transformed into a solution for the approximate L_2-post-office problem.

Let $p \in \mathbb{R}^2$ be a query point, and let p^* and p^∞ be the Euclidean neighbor and L_∞-neighbor of p, respectively. Let us take p^∞ as an approximate L_2-neighbor of p. How large is the error? That is, what is the largest value the quotient $d_2(p,p^\infty)/d_2(p,p^*)$ can have?

Exercise 1 Prove that in the L_∞-metric, a circle centered at p and having radius one is an axes-parallel square with sides of length two that is centered at p.

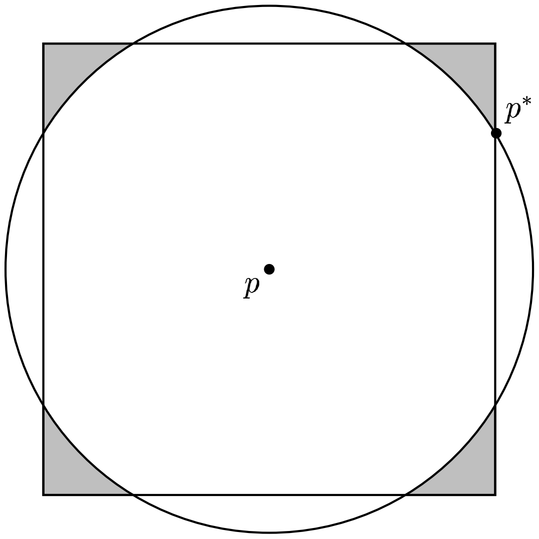

Figure 1: p^* *lies on the boundary of the square centered at* p *having sides of length* 2δ*, where* $\delta = d_\infty(p, p^*)$*. The circle is centered at* p *and has radius* $d_2(p, p^*)$*. The* L_∞*-neighbor* p^∞ *lies in the shaded region.*

We can visualize the process of finding p^* and p^∞ as follows. To find p^*, we grow a circle centered at p until its boundary hits a point of S. This point is the Euclidean neighbor p^* of p. Similarly, to find p^∞, we grow an L_∞-circle, which is an axes-parallel square, centered at p until its boundary hits a point. This point is the L_∞-neighbor p^∞ of p.

This observation allows us to bound the quotient $d_2(p, p^\infty)/d_2(p, p^*)$. Let $\delta := d_\infty(p, p^*)$, and consider the axes-parallel square C with sides of length 2δ that is centered at p. See Figure 1. Note that p^* lies on the boundary of C. Since p^∞ is the L_∞-neighbor of p, it must lie inside or on the boundary of C. Hence, $d_2(p, p^\infty) \leq \sqrt{2} \cdot \delta$, and equality occurs if and only if p^∞ coincides with one of the corners of C. Similarly, $d_2(p, p^*) \geq \delta$, and equality occurs if and only if p^* coincides with the midpoint of one of the four sides of C. It follows that

$$d_2(p, p^\infty) \leq \sqrt{2} \cdot \delta \leq \sqrt{2} \cdot d_2(p, p^*).$$

Hence, p^∞ is a $\sqrt{2}$-approximate L_2-neighbor of p. The error is maximum if and only if p^* is the midpoint of a side and p^∞ is a corner of C.

Now consider the circle with center p and radius $d_2(p, p^*)$. The point p^∞ lies outside or on the boundary of this circle. Hence, p^∞ must lie in the shaded region of Figure 1.

We see from Figure 1 that the upper bound on the error depends on the angle between the line segment pp^* and the X-axis: The error can be maximum, i.e., $\sqrt{2}$, only if this angle is $0, \pi/2, \pi$, or $3\pi/2$. On the other hand, if the angle is close to $\pi/4, 3\pi/4, 5\pi/4$, or $7\pi/4$, the shaded region in Figure 1 is very small, and the quotient $d_2(p, p^\infty)/d_2(p, p^*)$ is close to one. In fact, if the angle is exactly $\pi/4, 3\pi/4, 5\pi/4$, or $7\pi/4$, then p^* and p^∞ are equal.

Exercise 2 Let α be the angle between the segment pp^* and the positive X-axis. Assume that $0 \leq \alpha \leq \pi/2$. Prove that

- $d_2(p, p^\infty) \leq \sqrt{2} \cdot d_2(p, p^*) \cos\alpha$ if $\alpha \leq \pi/4$, and
- $d_2(p, p^\infty) \leq \sqrt{2} \cdot d_2(p, p^*) \sin\alpha$ if $\alpha \geq \pi/4$.

Here is the conclusion: The L_∞-neighbor p^∞ is a good approximation for the L_2-neighbor p^* if the angle between pp^* and the positive X-axis is close to $\pi/4, 3\pi/4, 5\pi/4$, or $7\pi/4$. Of course, in general, this angle will not be close to any of these four values. Nevertheless, we can use this approach to find a $(1+\varepsilon)$-approximate L_2-neighbor of p. The idea is to maintain a number of different coordinate systems such that there is always at least one system in which the angle between pp^* and its X-axis is close to $\pi/4$.

The details are as follows. We assume that the (XY)-coordinate system is given. Let $0 < \varepsilon \leq \pi/4$. For each i, $0 \leq i < 2\pi/\varepsilon$, let X_i and Y_i be the directed lines that make angles of $i \cdot \varepsilon$ with the positive X- and Y-axis, respectively. Consider the (X_iY_i)-coordinate systems, $0 \leq i < 2\pi/\varepsilon$.

Lemma 1 *For each point q in the plane, there is an index i, such that the angle between the line segment from the origin to q and the positive X_i-axis lies in between $\pi/4$ and $\pi/4 + \varepsilon$.*

Proof: Let $\vec{q}$ be the line segment from the origin to q, and let γ be the angle between $\vec{q}$ and the positive X-axis. First assume that $\pi/4 \leq \gamma < 2\pi$. Let $i := \lfloor(\gamma - \pi/4)/\varepsilon\rfloor$ and let γ_i be the angle between $\vec{q}$ and the positive X_i-axis. Then $\gamma_i = \gamma - i \cdot \varepsilon$ and $\pi/4 \leq \gamma_i \leq \pi/4 + \varepsilon$.

If $0 \leq \gamma < \pi/4$, then we take $i := \lfloor(7\pi/4 + \gamma)/\varepsilon\rfloor$. In this case, $\gamma_i = 2\pi - i \cdot \varepsilon + \gamma$ and again $\pi/4 \leq \gamma_i \leq \pi/4 + \varepsilon$. ■

For each i, $0 \leq i < 2\pi/\varepsilon$, let S_i denote the set of points in S with coordinates in the (X_iY_i)-coordinate system. Let p be a query point and let $q^{(i)}$ be an L_∞-neighbor of p in S_i, $0 \leq i < 2\pi/\varepsilon$. Let q be the L_∞-neighbor having minimum L_2-distance to p.

Lemma 2 *The point q is a $(1+\varepsilon)$-approximate L_2-neighbor of p.*

Proof: First note that the L_∞-distance depends on the coordinate system. Each (X_iY_i)-system has its own L_∞-distance function. The L_2-distance, however, is the same in all these systems. Let p^* be the exact L_2-neighbor of p. We have to show that

$$d_2(p,q) \leq (1+\varepsilon) \cdot d_2(p,p^*).$$

By Lemma 1, there is an index i such that the angle γ_i between the line segment from the origin to the point $p^* - p$ and the positive X_i-axis satisfies $\pi/4 \leq \gamma_i \leq \pi/4 + \varepsilon \leq \pi/2$. Note that γ_i is also the angle between the line segment from p to p^* and the positive X_i-axis. Exercise 2 implies that

$$\begin{aligned} d_2(p,q^{(i)}) &\leq \sqrt{2} \cdot d_2(p,p^*) \sin \gamma_i \\ &\leq \sqrt{2} \cdot d_2(p,p^*) \sin(\pi/4 + \varepsilon) \\ &= (\cos \varepsilon + \sin \varepsilon) d_2(p,p^*), \end{aligned}$$

where we used the formula $\sin(\alpha + \beta) = \sin\alpha \cos\beta + \cos\alpha \sin\beta$. Since $0 < \varepsilon \leq \pi/4$, we have $0 < \cos\varepsilon \leq 1$ and $0 < \sin\varepsilon \leq \varepsilon$. Therefore,

$$d_2(p,q^{(i)}) \leq (1+\varepsilon) \cdot d_2(p,p^*).$$

Now consider the point q. This point has minimum L_2-distance to p among all L_∞-neighbors $q^{(j)}$, $0 \leq j < 2\pi/\varepsilon$. In particular, $d_2(p,q) \leq d_2(p,q^{(i)})$. This completes the proof. ∎

Remark 1 We showed that $d_2(p,q)/d_2(p,p^*) \leq 1+\varepsilon$. By a more careful analysis, it can be shown that in fact $d_2(p,q)/d_2(p,p^*) \leq \sqrt{2}\cos(\pi/4 - \varepsilon/2)$.

We have proved that any solution for the exact L_∞-post-office problem can be used to solve the approximate L_2-post-office problem:

Theorem 1 *Let $\varepsilon > 0$ be a constant. The complexity of the planar $(1+\varepsilon)$-approximate L_2-post-office problem is at most $O(1/\varepsilon)$ times the complexity of the planar exact L_∞-post-office problem.*

In Section 3, we will see how the L_∞-post-office problem can be solved using range trees. This data structure is introduced in the next section.

2 Range trees

Range trees are based on balanced binary search trees. We use binary trees as *leaf search trees*: Let V be a finite subset of $\mathbb{R} \cup \{-\infty, \infty\}$, and let n be the size of V. We assume that V contains the elements $-\infty$ and ∞. A leaf search tree for V is a binary tree storing the elements of V in its leaves, sorted from left to right. Internal nodes contain information to guide searches. That is, each internal node u stores the values

1. $maxl(u)$, which is the maximum value stored in the left subtree of u, and

2. $minr(u)$, which is the minimum value stored in the right subtree of u.

Exercise 3 (1) Prove that any leaf search tree for V consists of $2n-1$ nodes. (2) Let $x \in \mathbb{R}$. Give an algorithm that finds the smallest element of V that is greater than or equal to x. Similarly, show how to find the largest element of V that is less than or equal to x.

Clearly, the best performance is obtained if the binary tree is *perfectly balanced*, i.e., for each internal node u, the number of leaves in the left and right subtrees of u differ by at most one. It is easy to see that such a tree has height $O(\log n)$.

Exercise 4 Give two algorithms, one bottom-up and the other top-down, to construct a perfectly balanced leaf search tree for V in $O(n \log n)$ time. If the elements of V are sorted already, the running time should be $O(n)$.

Exercise 5 Give an exact formula (a function of n) for the height of a perfectly balanced leaf search tree for V.

We are now ready to define the range tree. Let S be a set of n points in the plane.

Assumption 1 *All x-coordinates of the points of S are distinct, and the same is true for the y-coordinates of the points of S.*

Hence, no two points of S lie on a horizontal or vertical line. This assumption is made to simplify the algorithm. Later, we will see how it can be removed.

Definition 1 A *range tree* for S consists of the following:

1. An *x-tree* (also called *main tree*), which is a perfectly balanced leaf search tree for the x-coordinates of the points of S and the artificial x-coordinates $-\infty$ and ∞.

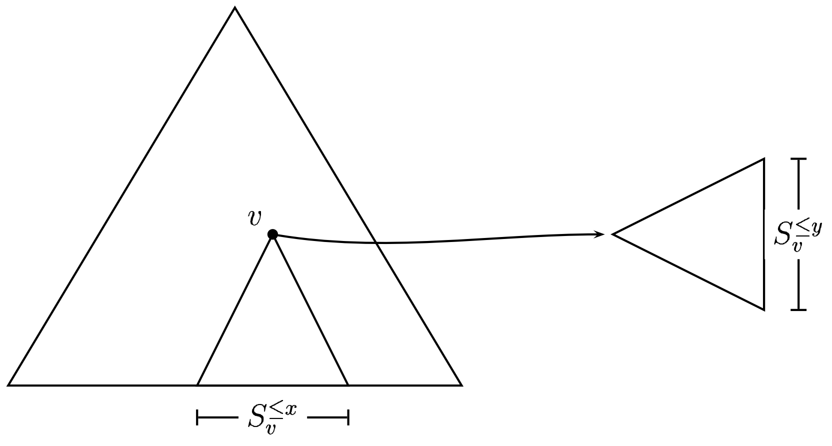

Figure 2: *A range tree. The subtree of each node v stores the x-coordinates of the points of S_v in sorted order in its leaves. The y-tree of v stores the y-coordinates of these points, and the values $-\infty$ and ∞, in sorted order in its leaves.*

---

2. Each node v of this tree contains a pointer to a *y-tree* (also called *associated* or *secondary structure*): Let S_v be the set of points of S whose x-coordinates are stored in the subtree of v. The y-tree of v is a perfectly balanced leaf search tree for the y-coordinates of the points of S_v and the artificial y-coordinates $-\infty$ and ∞.

See Figure 2 for a pictorial representation of a range tree. Note that S_v is a subset of S. In particular, if v is the rightmost leaf of the x-tree, then $S_v = \emptyset$, although v stores the artificial x-coordinate ∞.

Of course in an implementation, we store with each x- and y-coordinate (a pointer to) the corresponding point of S. Consider a node v of the x-tree. Then we can search in the set S_v for an x-coordinate as well as for a y-coordinate. This makes range trees useful for solving geometric problems.

Let p be a point in the plane. Often we want to search (e.g. with a y-coordinate or a range of y-coordinates) in the set of all points of S that are on or to the right of the vertical line through p. Using the x-tree, we can decompose this set into $O(\log n)$ pairwise disjoint subsets, as follows. Search in the x-tree for the smallest x-coordinate that is greater than or equal to the x-coordinate of p. During this search, each time we move from a node v to its left child, add the right child of v to an initially empty set M. The leaf in which the search ends is also added to M. See Figures 3 and 4. In

```
Algorithm decompose(p)
(* $p = (p_x, p_y)$ is a point in the plane *)
$M := \emptyset$;
$v :=$ root of the $x$-tree;
while $v \neq$ leaf
do if $maxl(v) < p_x$
    then $v :=$ right child of $v$
    else $M := M \cup \{$right child of $v\}$;
         $v :=$ left child of $v$
    endif
endwhile;
$M := M \cup \{v\}$
```

Figure 3: *Partitioning all points of S that are to the right of p into $O(\log n)$ subsets.*

Lemma 3 below, we will show that

$$\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u.$$

Hence, we indeed decompose the set of all points of S that are to the right of p into $O(\log n)$ subsets. If we want to search in this set (e.g. with a y-coordinate), then we can search in each set S_u, $u \in M$, separately.

Lemma 3 *Consider the set M of nodes of the x-tree that is computed by algorithm decompose(p). We have*

$$\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u,$$

and the right-hand side is a union of pairwise disjoint sets. The set M consists of $O(\log n)$ nodes.

Proof: Let v be the leaf in which algorithm *decompose*(p) ends, and let r be the point whose x-coordinate is stored in v. (We assume that v is not the rightmost leaf of the x-tree. In that case, the lemma is true.) Then $r_x \geq p_x$. All leaves in the subtree of any node $u \in M \setminus \{v\}$ are to the right of v. Hence, the x-coordinates stored in these leaves are greater than or equal to p_x. This proves that $\bigcup_{u \in M} S_u \subseteq \{q \in S : q_x \geq p_x\}$.

To prove the converse, let q be a point of S such that $q_x \geq p_x$. Let ℓ be the leaf that contains q_x. Then $\ell = v$, or ℓ is to the right of v. If $\ell = v$,

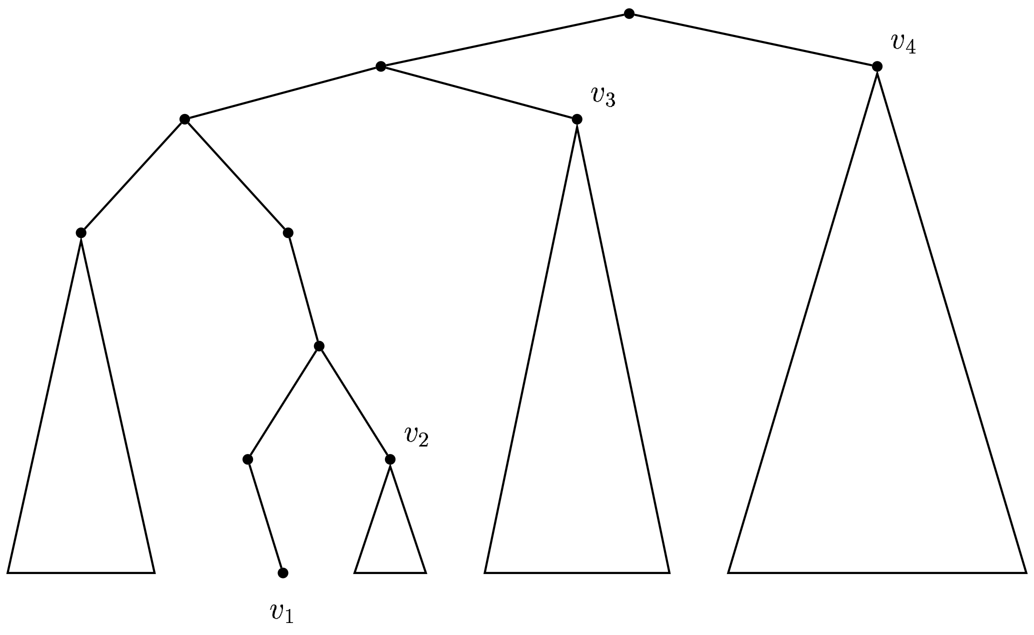

Figure 4: *The search for* p_x *ends in the leaf* v_1. *We have* $M = \{v_1, v_2, v_3, v_4\}$.

then $q \in \bigcup_{u \in M} S_u$, because $v \in M$. Assume $\ell \neq v$. Let w be the lowest common ancestor of v and ℓ. Then v is in the left subtree of w, ℓ is in the right subtree of w, w is on the search path to p_x, and in w this path moves to the left child of w. Therefore, the right child w' of w is contained in M. Since q_x is stored in the subtree of w', we have $q \in \bigcup_{u \in M} S_u$.

Next we prove that the sets S_u, $u \in M$, are pairwise disjoint. Let u and u' be two distinct nodes of M. First note that u is not contained in the subtree of u', and u' is not contained in the subtree of u. Let w be the lowest common ancestor of u and u'. Then, u and u' are contained in different subtrees of w. This proves that $S_u \cap S_{u'} = \emptyset$.

Since each node on the search path "delivers" at most one node to the set M, it follows that this set has size $O(\log n)$. ■

Let us analyze the size of a range tree for a set S of n points. Consider a fixed level of the x-tree, and let $u_1, u_2, \ldots, u_k$ be the nodes on this level. For each i, $1 \leq i \leq k$, the y-tree of u_i has size $O(|S_{u_i}|)$. Note that the sets S_{u_i}, $1 \leq i \leq k$, partition S. Therefore, $\sum_{i=1}^{k} |S_{u_i}| = n$. It follows that the y-trees of the nodes $u_1, u_2, \ldots, u_k$ together have size $O(n)$. This holds for any level of the x-tree. Hence, all y-trees together have size $O(n \log n)$. Since the x-tree itself has size $O(n)$, we have proved:

Lemma 4 *A range tree for any set of n points in the plane has size $O(n \log n)$.*

To finish this section, we consider the problem of building a range tree. It is clear that this takes $\Omega(n \log n)$ time. (Why?) Here is an algorithm that builds the data structure in $O(n \log n)$ time.

Step 1: Build the x-tree.

Step 2: Do the following for each leaf u of the x-tree: Let p be the point whose x-coordinate is stored in u. Give u a pointer to a y-tree storing the set $\{-\infty, p_y, \infty\}$. (The y-trees of the leftmost and rightmost leaves store the sets $\{-\infty, \infty\}$.)

Step 3: Build the y-trees of the internal nodes of the x-tree in a bottom-up fashion: If u is an internal node with children v and w, such that the y-trees of v and w have been built already, then we copy and merge these y-trees. (The values $-\infty$ and ∞ are stored only once in the resulting tree.) The tree obtained in this way is the y-tree of u.

Exercise 6 Prove that this algorithm builds a range tree in $O(n \log n)$ time.

We now explain how to remove Assumption 1. In the x-tree, we store the points using the lexicographical ordering instead of the ordering by x-coordinates. The search information stored in the internal nodes become points instead of x-coordinates. Similarly, in a y-tree, we store points using the "reversed" lexicographical ordering (a y-coordinate has higher priority than an x-coordinate). The algorithms are only slightly changed. In algorithm *decompose*(p), we search for the leftmost leaf that stores a point whose x-coordinate is greater than or equal to p_x.

3 Solving the L_∞-post-office problem

Recall the problem we want to solve: Preprocess a set S of n planar points in a data structure, such that for any query point $p \in \mathbb{R}^2$, we can find its L_∞-neighbor, i.e., a point $p^\infty \in S$ such that

$$d_\infty(p, p^\infty) = \min\{d_\infty(p, q) : q \in S\}.$$

We will show that this problem can be solved using range trees.

Consider a query point p. Let p^l and p^r be the L_∞-neighbors of p in the sets $\{q \in S : q_x \leq p_x\}$ and $\{q \in S : q_x \geq p_x\}$, respectively. We call these points the *left-L_∞-neighbor* and *right-L_∞-neighbor* of p, respectively. Clearly, one of them is the L_∞-neighbor of p.

We will show how to find the right-L_∞-neighbor p^r of p. (This point may not be unique. Actually, we should talk about *a* right-L_∞-neighbor.) The algorithm consists of three stages. Here is a brief overview.

Stage 1: Run algorithm *decompose*(p), see Figure 3. This algorithm computes a set M of nodes such that

$$\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u.$$

Number these nodes $v_1, v_2, \ldots, v_m$, where $m = |M|$, and v_i is closer to the root than v_{i-1}, $2 \leq i \leq m$. (See Figure 4.)

Stage 2: We know that the right-L_∞-neighbor p^r is contained in the union $\bigcup_{u \in M} S_u$. In the second stage, we want to search for a node $v \in M$ such that S_v contains p^r. This turns out to be difficult. We can, however, reach the following weaker goal: We compute a node $v \in M$ and a "small" set $C \subseteq S$ such that $C \cup S_v$ contains p^r.

Stage 3: Given node v and set C from stage 2, we walk down the subtree of v. During this walk, we maintain the invariant that $C \cup S_v$ contains p^r. If v is a leaf, then the set $C \cup S_v$ is small enough to look at all its points and take the one having minimum L_∞-distance to p. This point is the right-L_∞-neighbor p^r of p.

We now discuss Stages 2 and 3 in more detail.

3.1 Stage 2

We run the algorithm given in Figure 5. In words, this algorithm does the following. It visits the nodes of M from left to right (or, equivalently, from bottom to top). Consider one iteration. (See Figure 6.) The algorithm searches with p_y in the y-tree of v_i. This gives two points a and b of S_{v_i} between which (w.r.t. the vertical direction) p lies. Let r be the point of S whose x-coordinate is stored in the rightmost leaf of the subtree of v_i. The vertical lines through p and r define a slab whose width is denoted by δ. Note that all points of S_{v_i} lie in or on the boundary of this slab. Consider the rectangle R. If a and b are both outside R, then we add these two points to C, and go to the next iteration. Otherwise, if a or b is in R or on its boundary, then the while-loop stops.

Remark 2 Since the x- and y-trees also store values $-\infty$ and ∞, we have to be careful. If the rightmost leaf in the subtree of v_i stores the value ∞, then

```
$C := \emptyset$; $i := 1$; $stop := false$;
while $i \leq m$ and $stop = false$
do search in the $y$-tree of $v_i$ for the largest and smallest
    $y$-coordinates that are less than and greater than
    or equal to $p_y$, respectively;
    let $a$ and $b$ be the points of $S$ that correspond to
    these $y$-coordinates;
    $r :=$ the point of $S$ whose $x$-coordinate is stored in
         the rightmost leaf of the subtree of $v_i$;
    $\delta := r_x - p_x$;
    $R :=$ the rectangle $[p_x : r_x] \times [p_y - \delta : p_y + \delta]$;
    if $a$ and $b$ are both outside $R$
    then $C := C \cup \{a, b\}$;
          $i := i + 1$
    else $v := v_i$;
          $stop := true$
    endif
endwhile
```

Figure 5: *Stage 2.*

there is no point r corresponding to it. In this case, the value of δ, which is $r_x - p_x = \infty - p_x$ according to the algorithm, is set to ∞. As a result, the rectangle R is the halfplane to the right of the vertical line through p.

Similarly, the y-coordinate b_y may be ∞. Then, there is no point b corresponding to this value. In this case, we use an artificial point b which is outside rectangle R if r_x is finite, and inside R if $r_x = \infty$. A y-coordinate $a_y = -\infty$ is treated in a similar way.

We consider the variable *stop* at the end of the while-loop, and consider the cases when this variable has value *true* or *false* separately.

Lemma 5 *If the variable stop has value false after the while-loop has been completed, then the set C contains a right-L_∞-neighbor of p.*

Proof: First note that the while-loop makes m iterations. Let p^r be a right-L_∞-neighbor of p, and let i be the index such that $p^r \in S_{v_i}$. Consider the i-th iteration of the while-loop. The points a and b selected during this iteration are outside R.

Let q be any point of S_{v_i}. Then $p_x \leq q_x \leq r_x$ and, hence, $0 \leq q_x - p_x \leq r_x - p_x = \delta$. On the other hand, since q is outside R, we have $|q_y - p_y| > \delta$. It

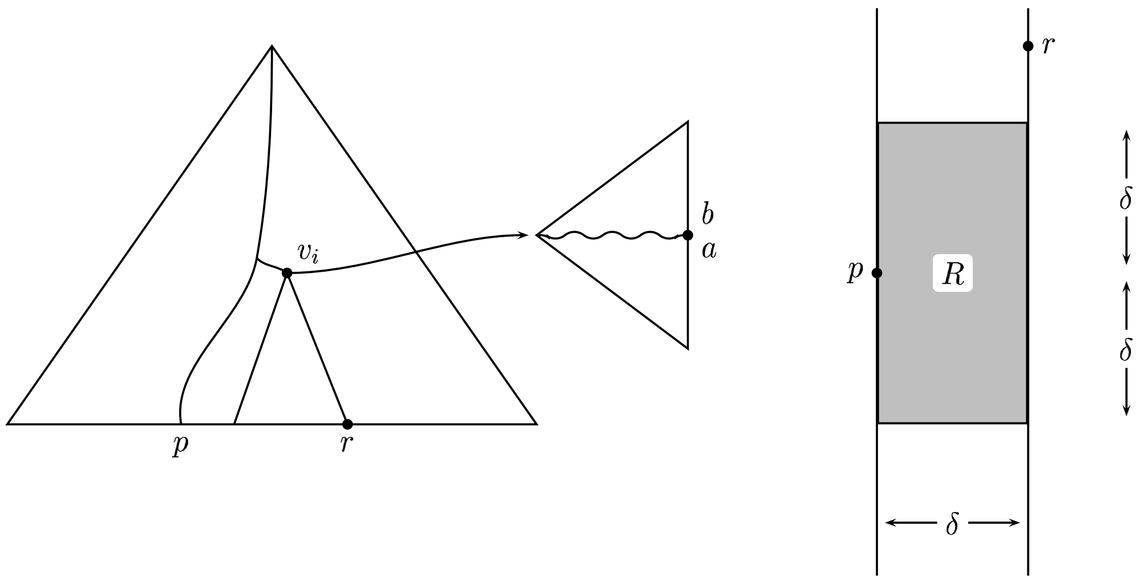

Figure 6: *Illustrating one iteration of Stage 2.*

follows that $d_\infty(p,q) = |q_y - p_y|$. That is, for all points of S_{v_i}, the L_∞-distance to p is the same as the distance to p in the y-direction.

Assume w.l.o.g. that $d_\infty(p,a) \leq d_\infty(p,b)$. Then

$$d_\infty(p,a) = |p_y - a_y| \leq |p_y - p^r_y| = d_\infty(p,p^r).$$

On the other hand, since p^r is a right-L_∞-neighbor of p, we have $d_\infty(p,p^r) \leq d_\infty(p,a)$. This proves that $d_\infty(p,a) = d_\infty(p,p^r)$.

Hence, a is also a right-L_∞-neighbor of p. Since a is added to C during the i-th iteration, the proof is completed. ∎

If the variable *stop* has value *false* at the end of the while-loop, then we can easily complete the algorithm: We consider all points of C and take the one having minimum L_∞-distance to p. By Lemma 5, this point is a right-L_∞-neighbor of p.

Lemma 6 *If the variable stop has value true after the while-loop has been completed, then the set $C \cup S_v$ contains a right-L_∞-neighbor of p.*

Proof: Let p^r be a right-L_∞-neighbor of p, and let i be the index such that $p^r \in S_{v_i}$. Let j be the integer such that during the j-th iteration of the while-loop, the variable *stop* is set to the value *true*. Note that $v = v_j$.

First assume that $i < j$. During the i-th iteration, the points a and b that are selected in the y-tree of v_i are outside the rectangle R. In exactly the same way as in the proof of Lemma 5, it follows that a or b is also a right-L_∞-neighbor of p. Since both points are added to C during the i-th iteration, the claim follows.

Next assume that $i = j$. Then the set S_v, hence also the set $C \cup S_v$, contains a right-L_∞-neighbor of p.

It remains to consider the case when $i > j$. Consider what happens during the j-th iteration. Let a and b be the points in the y-tree of v_j that are selected during this iteration. At least one of them is contained in the rectangle R. Assume w.l.o.g. that a is in R. Then $d_\infty(p, a) \leq \delta$, where δ is the x-distance between p and the rightmost point r in the subtree of v_j.

Since the x-coordinates of all points in S_{v_j} are less than or equal to the x-coordinates of the points in S_{v_i}, we have $p^r_x \geq r_x$. This implies that

$$d_\infty(p, p^r) \geq p^r_x - p_x \geq r_x - p_x = \delta.$$

We have shown that $d_\infty(p, a) \leq d_\infty(p, p^r)$. On the other hand, since p^r is a right-L_∞-neighbor of p, we have $d_\infty(p, p^r) \leq d_\infty(p, a)$. This proves that $d_\infty(p, p^r) = d_\infty(p, a)$ and, hence, a is also a right-L_∞-neighbor of p. Since $a \in S_{v_j} = S_v$, the proof of the lemma is completed. ■

This concludes Stage 2. To summarize, if the variable *stop* has value *false* after the while-loop has been completed, then we find a right-L_∞-neighbor of p by considering all points of C. In this case, the algorithm terminates. Otherwise, we know that the set $C \cup S_v$ contains a right-L_∞-neighbor of p. In this case, we proceed to the next stage.

3.2 Stage 3

We run the algorithm given in Figure 7.

Lemma 7 *During the while-loop, the set $C \cup S_v$ contains a right-L_∞-neighbor of p.*

Exercise 7 Prove Lemma 7.

We complete Stage 3 as follows. Note that at this moment, v is a leaf of the x-tree. By considering all points of $C \cup S_v$, we take the one having minimum L_∞-distance to p. By Lemma 7, this point is a right-L_∞-neighbor of p.

```
while $v$ is not a leaf
do $w$ := left child of $v$;
    search in the $y$-tree of $w$ for the largest and smallest
    $y$-coordinates that are less than and greater than
    or equal to $p_y$, respectively;
    let $a$ and $b$ be the points of $S$ that correspond to
    these $y$-coordinates;
    $r$ := the point of $S$ whose $x$-coordinate is stored in
        the rightmost leaf of the subtree of $w$;
    $\delta := r_x - p_x$;
    $R$ := the rectangle $[p_x : r_x] \times [p_y - \delta : p_y + \delta]$;
    if $a$ and $b$ outside $R$
    then $C := C \cup \{a, b\}$;
        $v$ := right child of $v$
    else $v := w$
    endif
endwhile
```

Figure 7: *Stage 3.*

This concludes the algorithm for computing a right-L_∞-neighbor p^r of p. In a completely symmetric way, we compute a left-L_∞-neighbor p^l of p. Then, if $d_\infty(p, p^l) \leq d_\infty(p, p^r)$, point p^l is an L_∞-neighbor of p. If $d_\infty(p, p^l) > d_\infty(p, p^r)$, then point p^r is an L_∞-neighbor of p.

We analyze the running time of the query algorithm. By Lemma 3, Stage 1 takes $O(\log n)$ time. Consider the while-loop of Stage 2. Each iteration takes $O(\log n)$ time. Since $m = O(\log n)$, there are $O(\log n)$ iterations. Therefore, the entire while-loop takes $O((\log n)^2)$ time. If the variable *stop* has the value *false* after this loop, then we need $O(|C|)$ time to find a right-L_∞-neighbor of p. It is clear that $|C| \leq 2|M|$. Hence, $|C| = O(\log n)$. This proves that Stage 2 takes $O((\log n)^2)$ time.

In the while-loop of Stage 3, we walk down a path in the subtree of v. In each node on this path, we spend $O(\log n)$ time. Since this path has length $O(\log n)$, the entire loop takes $O((\log n)^2)$ time. Afterwards, we need $O(|C \cup S_v|)$ time to find a right-L_∞-neighbor. Since v is a leaf at this moment, we have $|S_v| = 1$. The size of C is bounded by $O(\log n)$. Therefore, this final step takes $O(\log n)$ time.

We have shown that the algorithm finds a right-L_∞-neighbor of p in $O((\log n)^2)$ time. In the same amount of time, a left-L_∞-neighbor is found.

Given these two points, the L_∞-neighbor of p is obtained in $O(1)$ time.

We summarize our result.

Theorem 2 *Let S be any set of n points in the plane. Using a range tree, which has size $O(n \log n)$, we can solve the L_∞-post-office-problem with a query time of $O((\log n)^2)$.*

Applying Theorem 1 gives:

Corollary 1 *Let $\varepsilon > 0$ be any constant, and let S be any set of n points in the plane. The $(1+\varepsilon)$-approximate L_2-post-office problem can be solved, using $O(n \log n)$ space, with a query time of $O((\log n)^2)$.*

4 Improving the query time: layering

We have seen that a range tree solves the two-dimensional L_∞-post-office problem with a query time of $O((\log n)^2)$. In this section, we reduce the query time to $O(\log n)$.

Consider the query algorithm of the previous section. This algorithm makes $O(\log n)$ binary searches in different y-trees, and it does some additional work. It is easily seen that the additional work takes only $O(\log n)$ time. (Here, we assume that we store with each node of the x-tree a pointer to the rightmost leaf in its subtree.) The $O(\log n)$ binary searches together take $O((\log n)^2)$ time. That is, the running time of the query algorithm is dominated by the time of these binary searches.

How can we improve the running time? The key observation is that in each y-tree, we search for the *same* element: we search for the y-coordinate of the query point p.

Let u and v be nodes of the x-tree such that u is a child of v. Assume we want to locate p_y in the y-trees of u and v. Recall that S_u and S_v denote the points of S that are stored in the subtrees of u and v, respectively.

Assume that the y-coordinate p_y is less than all y-coordinates of the points of S_v. Then the search for the smallest element in the y-tree of v that is greater than or equal to p_y will end in the second leftmost leaf of this y-tree. (The leftmost leaf stores the artificial y-coordinate $-\infty$.) Where does the search in the y-tree of u end? Since $S_u \subseteq S_v$, it is clear that p_y is smaller than all y-coordinates of the points of S_u. Therefore, the search for the smallest element in the y-tree of u that is greater than or equal to p_y also ends in the second leftmost leaf of this y-tree.

In general, the search for p_y in the y-tree of v gives information about the result of a search for the same element in the y-tree of u. As we will see,

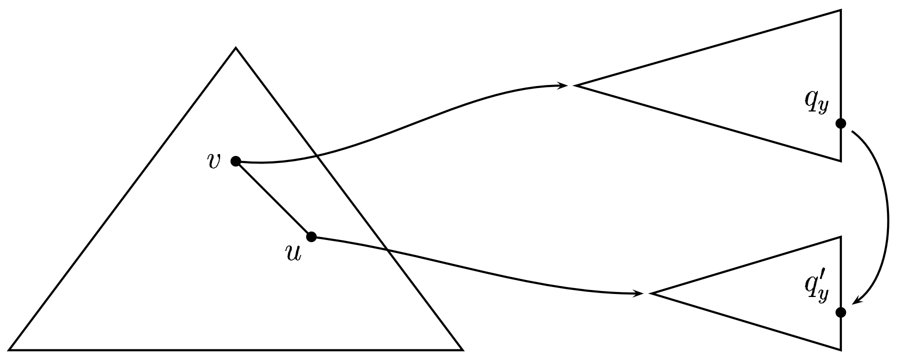

Figure 8: *A layered range tree. The leaf storing q_y contains a pointer to the leaf storing q'_y. If $q_y = -\infty$, then $q'_y = -\infty$. If $q_y = \infty$, the $q'_y = \infty$. If q_y is finite, then $q'_y = \min\{s_y : s = (s_x, s_y) \in S_u, s_y \geq q_y\}$.*

---

we can use this information such that, given the position of p_y in the y-tree of v, only $O(1)$ time is needed to locate p_y in the y-tree of u. That is, we avoid making a binary search in this y-tree. The idea is to link the y-trees of u and v by pointers. This technique is called *layering.*

Hence, we change the range tree as follows:

1. As before, we have an x-tree which is a perfectly balanced leaf search tree for the x-coordinates of the points of S and the artificial x-coordinates $-\infty$ and ∞.

2. Each node v of the x-tree contains a pointer to a y-tree, which is a perfectly balanced leaf search tree for the y-coordinates of the points of S_v and the artificial y-coordinates $-\infty$ and ∞.

3. For all nodes u and v of the x-tree, such that u is a child of v, there are pointers from the y-tree of v to the y-tree of u: Let ℓ be any leaf in the y-tree of v, and let q_y be the y-coordinate stored in ℓ. Leaf ℓ stores a pointer to the leftmost leaf in the y-tree of u whose y-coordinate is greater than or equal to q_y.

We call the resulting data structure a *layered range tree.* See Figure 8. Note that if q_y also occurs as a y-coordinate of a point in S_u, then the pointer from ℓ points to the occurrence of q_y in the y-tree of u.

Exercise 8 Prove that a layered range tree still has size $O(n \log n)$ and that it can be built in $O(n \log n)$ time.

Let p be any point in the plane. Run algorithm *decompose*(p). (See Figure 3.) This gives a set M of nodes of the x-tree such that $\{q \in S : q_x \geq p_x\} = \bigcup_{u \in M} S_u$. We show how to search for the smallest element that is greater than or equal to p_y, in the y-tree of each node u of M.

We again walk down the path in the x-tree to the leftmost leaf whose point has an x-coordinate greater than or equal to p_x. (Hence, we again follow the path that has been computed by algorithm *decompose*(p).) This path starts in the root v of the x-tree. We locate p_y in the y-tree of v. Let w be the right child of v. Then, by following the pointer from the leaf in v's y-tree that stores the position of p_y, to the y-tree of w, we have located p_y in w's y-tree. (See Lemma 8 below.) If w is on the path to p_x, then we proceed in the subtree of w. Otherwise, let u be the left child of v. Note that $w \in M$. We follow the pointer from the leaf in v's y-tree that stores the position of p_y, to the y-tree of u. This gives the smallest y-coordinate in the y-tree of u that is greater than or equal to p_y. Now we proceed in the subtree of u. The complete algorithm, which is denoted by *searchM*(p), is given in Figure 9.

Lemma 8 *During the while-loop of algorithm searchM*(p)*, the invariant, which is mentioned in the pseudo code, is correctly maintained.*

Proof: It is clear that the invariant holds after the initialization. Consider one iteration. That is, let v be a node of the x-tree, let ℓ be a leaf in the y-tree of v, let q be the point of S whose y-coordinate is stored in ℓ, and assume that

$$q_y = \min\{s_y : s_y \geq p_y \text{ and } s_y \text{ stored in the } y\text{-tree of } v\}. \tag{1}$$

Let w be the right child of v, let ℓ' be the leaf in the y-tree of w that is reached by following the pointer stored with ℓ, and let q' be the point of S whose y-coordinate is stored in ℓ'. We will show that

$$q'_y = \min\{s_y : s_y \geq p_y \text{ and } s_y \text{ stored in the } y\text{-tree of } w\}.$$

From the definition of the pointers that link the y-tree of v with that of w, we know that $q'_y \geq q_y$. Since $q_y \geq p_y$, we infer that

$$q'_y \in \{s_y : s_y \geq p_y \text{ and } s_y \text{ stored in the } y\text{-tree of } w\}.$$

It remains to show that q'_y is the minimum element of this set. Assume this is not the case. Then there is a y-coordinate r_y stored in the y-tree of w such that $p_y \leq r_y < q'_y$. Note that $S_w \subseteq S_v$. Therefore, the y-coordinates stored in the y-tree of w form a subset of those stored in the y-tree of v. In particular, r_y is stored in the y-tree of v. Since $r_y \geq p_y$, we infer from (1) that $r_y \geq q_y$.

We have shown that $q_y \leq r_y < q'_y$, where r_y is stored in the y-tree of w. But then, the pointer from ℓ in the y-tree of v cannot point to the leaf ℓ' storing q'_y. This is a contradiction.

This shows the q'_y is the smallest value in the y-tree of w that is greater than or equal to p_y. If the search path proceeds to w, then the invariant still holds after this iteration. Otherwise, if the search path proceeds to the left child u of v, then it follows in the same way that q''_y (see Figure 9) is the smallest element in the y-tree of u that is greater than or equal to p_y. Hence, also in this case, the invariant still holds after this iteration. This completes the proof. ∎

Lemma 9 *Each pointer that is reported by algorithm searchM(p) points to the leftmost leaf in the y-tree of a node of M, whose y-coordinate is greater than or equal to p_y.*

Proof: This follows immediately from the previous proof. ∎

We analyze the running time of algorithm *searchM*(p). The initialization takes $O(\log n)$ time. It is easy to see, that each iteration of the while-loop takes $O(1)$ time. Since there are $O(\log n)$ iterations, the entire while-loop takes $O(\log n)$ time. This proves that the entire algorithm runs in $O(\log n)$ time. That is, by introducing the layered range tree, we reduced the time to locate p_y in the y-trees of all nodes of M, from $O((\log n)^2)$ to $O(\log n)$.

Now we return to the algorithm of Section 3 for finding an L_∞-neighbor of a query point. We replace Stage 1 by algorithm *searchM*(p). Then, Stage 2 can be performed in $O(\log n)$ time. In a similar way, the running time for Stage 3 becomes $O(\log n)$. This proves:

Theorem 3 *Let S be any set of n points in the plane. Using a layered range tree, which has size $O(n \log n)$, we can solve the L_∞-post-office problem with a query time of $O(\log n)$.*

Corollary 2 *Let $\varepsilon > 0$ be any constant and let S be any set of n points in the plane. The $(1+\varepsilon)$-approximate L_2-post-office problem can be solved, using $O(n \log n)$ space, with a query time of $O(\log n)$.*

By generalizing range trees to D dimensions, where $D \geq 2$ is a constant, the following results can be proved:

Theorem 4 *Let S be any set of n points in $\mathbb{R}^D$, where $D \geq 2$ is a constant. Using a layered range tree, which has size $O(n(\log n)^{D-1})$, we can solve the L_∞-post-office problem with a query time of $O((\log n)^{D-1})$.*

Corollary 3 *Let $\varepsilon > 0$ be any constant and let S be any set of n points in $\mathbb{R}^D$, where $D \geq 2$ is a constant. The $(1+\varepsilon)$-approximate L_2-post-office problem can be solved, using $O(n(\log n)^{D-1})$ space, with a query time of $O((\log n)^{D-1})$.*

```
**Algorithm** *searchM*$(p)$
(* $p = (p_x, p_y)$ is a point in the plane *)
$M := \emptyset$;
$v :=$ root of the $x$-tree;
search in the $y$-tree of $v$ for the smallest $y$-coordinate that is
greater than or equal to $p_y$;
$\ell :=$ leaf where this search ends;
$q :=$ point of $S$ whose $y$-coordinate is stored in $\ell$;
while $v \neq$ leaf
do (* invariant: $\ell$ is a leaf in the $y$-tree of $v$, $\ell$ stores $q_y$,
      $q_y = \min\{s_y : s_y \geq p_y$ and $s_y$ stored in the $y$-tree of $v\}$ *)
   $w :=$ right child of $v$;
   follow the pointer from $\ell$ to the leaf $\ell'$ in the $y$-tree of $w$;
   $q' :=$ point of $S$ whose $y$-coordinate is stored in $\ell'$;
   if $maxl(v) < p_x$
   then $v := w$; $\ell := \ell'$; $q := q'$
   else $M := M \cup \{w\}$;
        output a pointer to $\ell'$;
        $u :=$ left child of $v$;
        follow the pointer from $\ell$ to the leaf $\ell''$ in the $y$-tree of $u$;
        $q'' :=$ point of $S$ whose $y$-coordinate is stored in $\ell''$;
        $v := u$; $\ell := \ell''$; $q := q''$
   endif
endwhile;
$M := M \cup \{v\}$
output a pointer to $\ell$
```

Figure 9: *Constructing the set M, and locating p_y in the y-trees of all nodes of M.*