Efficient Regular Data Structures and Algorithms for Dilation, Location and Proximity Problems

Arnon Amir* Alon Efrat† Piotr Indyk‡ Hanan Samet§

Abstract

In this paper we investigate data-structures obtained by a recursive partitioning of the input domain into regions of *equal* size. One of the most well known examples of such a structure is the *quadtree*, used here as a basis for more complex data structures; we also provide multidimensional versions of the *stratified tree* by van Emde Boas [20]. We show that under the assumption that the input points have limited precision (i.e. are drawn from the integer grid of size u) these data structures yield efficient solutions to many important problems. In particular, they allow us to achieve $O(\log \log u)$ time per operation for dynamic approximate nearest neighbor (under insertions and deletions) and exact on-line closest pair (under insertions only) in any constant dimension. They allow $O(\log \log u)$ point location in a given planar shape or in its expansion (dilation by a ball of a given radius). Finally, we provide a linear time (optimal) algorithm for computing the expansion of a shape represented by a quadtree. This result shows that the spatial order imposed by this regular data structure is sufficient to optimize the dilation by a ball operation.

1 Introduction

*IBM Almaden Research Center, 650 Harry Rd., San Jose, CA 95120. *arnon@almaden.ibm.com*

†Computer Science Department, Stanford University, *alon@cs.stanford.edu* , whose work is supported in part by Rothschild Fellowship and by DARPA contract DAAE07-98-C-L027

‡Computer Science Department, Stanford University, *indyk@cs.stanford.edu*

§Computer Science Department, University of Maryland, College Park, Maryland 20742. *hjs@umiacs.umd.edu*, whose work is supported in part by the National Science Foundation under grant IRI-97-12715 and the Department of Energy under Contract DEFG0295ER25237.

In this paper we consider spatial data structures which are based on (possibly recursive) decomposition of a bounded region into blocks, where the resulting blocks of each partition are of equal size; we call such structures *regular*. One of the most popular examples of such structures is the quadtree (e.g., [17, 18]), which is based on a recursive decomposition of a square into four quadrants; another example is the stratified tree structure by van Emde Boas [20]. The quadtree data structure and its numerous variants are one of the most widely used data structures for spatial data processing, computer graphics, GIS etc. Some of the reasons for this state of art are:

- Simplicity: the data structures and related algorithms are relatively simple and easy to implement
- Efficiency: they require much less storage to represent a shape than the full bit map
- Versatility: many operations on such data structures can be done very efficiently (for example computing the union/intersection or connected component labeling [5, 18])

Despite their usefulness, however, regular data structures for geometric problems have not been much investigated from the theoretical point of view. One of the main reasons is that in the widely adopted input model where the points are allowed to have arbitrary real coordinates, the depth and size of (say) a quadtree could be unbounded, thus making worst-case analysis of related algorithms impossible. On the other hand, such a situation is rarely observed in practice. One reason is that in many practical applications the coordinates are represented with fixed precision, thus making the unbounded size scenario impossible. Another reason is that the regions encountered in practice are not worst-case shapes; for example, they are often composed of fat objects[1]. It is therefore important to study such cases, for both practical and theoretical purposes.

In this paper we give a solid theoretical background to these cases. First, we prove that if the input precision is limited to say b bits (or alternatively, the input coordinates are integers from the interval $[u] = \{0 \dots u-1\}$ where $u = 2^b$), then by using regular data structures several location and proximity problems can be solved very efficiently. We also show that if the input shapes are unions of fat objects, then the space used by these data structures and their construction time is small.

[1] Later in this paper we provide the mathematical definitions for all following terms

Our first sequence of results apply to problems on sets of *points*. For this case, we propose two multidimensional generalizations of the stratified tree. The 1-dimensional version, by van Emde Boas [20], yields a dynamic data structure for nearest-neighbor queries. The running time which it guarantees, $O(\log\log u)$, has been recently improved to $O(\log\log u/\log\log\log u)$ (see [2]). We are not aware, however, of any prior work in which its multidimensional version has been applied.

The first multidimensional stratified tree allows $O(\log\log u)$ time per operation for dynamic approximate nearest neighbor (when insertions and deletions of points are allowed) and maintain the (exact) on-line closest pair when insertions are allowed. The result holds for any fixed dimension (the dependence on the dimension, however, is exponential). The data structure is randomized and the bounds hold in the expected sense.

The second multidimensional data structure is deterministic and static. It enables to answer approximate nearest neighbor query in d-dimensions in time $O(d + \log\log u)$ and space $d^{\log\log\log u} O(1/\epsilon)^d n \log^{O(1)} u$. Recently, Beame and Fich [2] showed a lower bound of $\Omega(\log\log u/\log\log\log u)$ for the case $d = 1$, assuming $n^{O(1)}$ storage[2]. Thus our algorithms is within factor $\log\log\log u$ from optimal as long as $d = O(\log n)$ (note that $O(d)$ is a trivial lower bound).

The remaining results apply to the case when the input shape is a union of objects which are more complex than points. In this case the stratified tree structure does not seem to suffice and therefore we resort to quadtrees. We consider two operations on shapes; expansion (dilation with a circle) and point location. Dilation and point-location are fundamental operations in various fields such as robotics, assembly, geographic information systems (GIS), computer vision and computer graphics. Thus having efficient algorithms for these problems is of major practical importance. Before describing these results in more details, we need the following definitions: Among the many variants of quadtrees exist in the literature, we consider the following types of quadtrees:

Region quadtree (or just quadtree): obtained by recursive subdivision of squares until each leaf is black/white (i.e. is inside/outside the shape).

[2] Although their proof works for the *exact* nearest neighbor, we show it generalizes to the *approximate* problem as well.

Segment (or *mixed*) quadtree: we allow the leaves to contain shapes of constant complexity $\leq \kappa$ (e.g. at most κ points)[3].

Compressed quadtree: a variant of either region quadtree or mixed quadtree, at which all sequences of adjacent nodes along a path of the tree having only one non-empty (i.e. which contain a part of the shape) child are compressed into one edge; An important property of compress quadtrees is that the number of nodes in the resulting tree, denoted as the *size* of the tree, is at most twice the number of its (non-empty) leaves.

Let S be a planar shape, and let r be a fixed radius. The *dilated shape* of S, denoted by $D(S)$, is the Minkowski sum of S with the disk D_r of radius r. That is, $D(S) = \{d + s \mid d \in D_r, s \in S\}$. In (Section 4) we provide an algorithm that takes a region quadtree of N nodes representing S and a radius r and computes $D(S)$ in optimal time $O(N)$.

In Section 5 we first address the efficiency of a region quadtree as a planer shape representation. It is well known that the size of a region quadtree could be much larger than the complexity of the shape S it represents. However, there are cases were a segment quadtree can be much more efficient. We show (Section 5.1) that if S can be expressed as a collection of n *fat* convex objects in $[u]^2$, then S can be represented as a segment quadtree with $N = O(|\partial S| \log u)$ leaves, where $|\partial S|$ is the complexity of the boundary of S, which in turn is known to be close to linear in n [7].

After we show that a segment quadtree can be an efficient shape representation, we proceed (Section 5.2) with an efficient algorithm to construct it. Given a decomposition of S into n (not necessarily fat) *disjoint* objects, the segment quadtree representing S can be constructed in time $O(N \log u)$, where N is the size of the output quadtree. It follows that $D(S)$ can be computed and stored in a segment quadtree in time $O(N \log^2 u)$, and as a compressed segment quadtree in time and space $O(N \log \log u)$.

In (Section 5.3) we provide an efficient point location algorithm for a shape S represented by a region quadtrees. Since the tree has depth $O(\log u)$, point location in S can easily be performed within $O(\log u)$ time. However, by performing a binary search on levels of the tree, one can reduce this time to $O(\log \log u)$,

[3]We assume that the boundary of an object can be expressed as a collection of algebraic arcs, connected in vertices. The number of these vertices is the *complexity* of the object.

with pre-processing time and space $O(N) = O(n \log u)$, where N and n are the number of nodes and leaves in the tree, respectively [21]. We show that for a *compressed* quadtree the query time is also $O(\log\log u)$ with pre-processing time/space $O(N \log\log u) = O(n \log\log u)$. Thus for the same shape S we reduce the pre-processing time and storage from $O(n \log u)$ to $O(n \log\log u)$.

Most of the algorithmic problems above have $\Omega(\log n)$ or $\Omega(n \log n)$ lower bounds in standard algebraic data model (assuming arbitrary precision input). Thus by resorting to a fixed precision model we are able to replace $\log n$ by $\log\log u$ in several run time bounds. Notice that in most situations this change yields a significant improvement. For example, when the numbers are represented using 32 bits then $\log\log u = 5$ while $\log n > 5$ already for $n > 32$. Moreover the regular data structures are usually much simpler than the corresponding solutions for the real data model, thus the "big-O" constants are likely to be smaller. Thus we expect our algorithms to yield better running times in practice, especially for scenarios when the input size is large.

There has been a number of papers discussing computational geometry problems on a grid. Examples include nearest neighbor searching using the L_1-norm [16], the point location problem [14] and orthogonal range searching [15]; the solutions given in these papers provide static data structures for two-dimensional data with query time $O(\log\log u)$. A number of off-line problems has been also considered (see [15] for more details). However, to our knowledge, there are no *dynamic* data structures known for grid with query/update times better than for arbitrary input; in fact, this is one of the open problems posed in [15, page 273].

2 Dynamic multidimensional stratified trees

In this section we present the multidimensional stratified tree (MDST), a multidimensional extension of the stratified tree. It addresses the dynamic approximate nearest neighbor problem in $[u]^d$. Let $P \subseteq [u]^d$ be a set of points, and let $\varepsilon > 0$ be a pre-specified parameter. Let $q \in [u]^d$ denote a query point, and let $p_{\mathrm{nn}} \in P$ denote the (exact) closest neighbor to q. We say that $p_{\mathrm{app}} \in P$ is ε approximate nearest neighbor of q if $d(q, p_{\mathrm{app}}) \le (1+\varepsilon) d(q, p_{\mathrm{nn}})$ (see, e.g. Arya et al, [1]). The dynamic data structure supports update operations (inserting or deleting a point) and approximate nearest neighbor queries in time $O(1/\epsilon^{O(d)} \log\log u)$ in $[u]^d$

(for $d \geq 2$). In addition, a simple reduction provides an algorithm to maintain *exact* closest pair (under insertions of new points) in the same time. For clarity, we describe the algorithm for the two dimensional case. The extension to higher dimensions is straightforward.

The one-dimensional space subdivision technique called *stratified tree* was proposed by van Emde Boas [20]. It supports nearest neighbor query in the integer interval $[u]$ in $O(\log\log u)$ time. The data structure can be made dynamic (cf. [12]). Each addition or deletion of a point takes $O(\log\log u)$ time. The tree requires $O(n)$ space, where n denotes the maximum number of points present in the tree at any time[4]. It is assumed that standard boolean and arithmetic operations on words of size $\log u$ can be performed in constant time; some of our results require this assumption.

The MDST, like the stratified tree, supports the following four procedures: *construct* (which constructs a tree for a given set of points), *add* and *delete* which enables addition/deletion of a point from the set and *search*, which finds an approximate nearest neighbor of a given query point. Below we give the description of the *construct* (together with the description of the data structure per se) and *search* procedures. The *add* and *delete* (nontrivial) procedures are essentially the same as in the one dimensional case, and the reader is therefore referred to [12] for details. In the rest of this section we first define the data structure and discuss the algorithm in general, including intuition behind this construction. Then we proceed with its correctness and complexity analysis.

The MDST data consists on three recursively coupled components, denoted D, E and H. Let $B \subset [2U]^2$ denote a *block*; the region corresponding to a node of the recursive construction. At the root the MDST starts with $B = [2U]^2$. A block B of size $v \times v$ is divided in the next level into v blocks, denoted B_{ij}, of size $\sqrt{v} \times \sqrt{v}$. Let $P \subset B$ denote a given set of points. The recursive procedure CONSTRUCT(B, P), shown in Figure 1, builds the MDST for the set of points P in the block B. Each node can be either a single-point leaf (case 0), a an internal node (case 1) or a bitmap leaf (case 2). Let us focus first on internal nodes. An internal node, illustrated in Figure 2, contains three components: an array D_{ij} of its v children, an array E_{ij} of v coarse (of size $\lambda \times \lambda$) binary maps, one for each of the children, and a bitmap H_{ij} of v bits that map

[4]The original paper by van Emde Boas provides an $O(u)$ space bound, but this can be reduced to $O(n)$ by using randomized dynamic hashing [21]; in such a case the time bounds are expected values.

the non-empty children. Note that by moving one level down along a path in the tree we determine half of the remaining bits of a point (e.g., compared with one bit in the case of a binary tree). Hence the maximal depth of the tree is $O(\log\log U)$. For any $p \in P \subset [v]^2$ we can quickly retrieve (after some preprocessing) *some* actual data set point belonging to the same sub-block containing p; we refer to such a point as an *actual* point of p. We use the MDST data structure for the construction of each of the three components of the node's data structure. Hence, all the three are recursively coupled together.

CONSTRUCT$([v]^2, P)$**:**

Let $\lambda = 1/\epsilon \cdot \log^2 u$, $v_0 = \lambda^{1/8}$

Case 0. $|P| = 1$: // *single-point leaf*

store the point p from P.

Case 1. $|P| > 1$ and $v > v_0$: // *internal node*

1. split $[v]^2$ into v square blocks B_{ij}, $i, j = 0 \ldots \sqrt{v} - 1$, each of size $\sqrt{v} \times \sqrt{v}$
2. for each B_{ij}

   $D_{ij} = \text{CONSTRUCT}(B_{ij}, P \cap B_{ij})$
3. if $\lambda < \sqrt{v}$ then for each B_{ij}
   (a) split B_{ij} into λ^2 square blocks C_{kl}, $k, l = 0 \ldots \lambda - 1$
   (b) compute the set $S_{ij} = \{(k,l) : C_{kl} \cap P \neq \emptyset\}$
   (c) $E_{ij} = \text{CONSTRUCT}([\lambda]^2, S_{ij})$
4. compute a set $H = \{(i,j) : B_{ij} \cap P \neq \emptyset\}$
5. $F = \text{CONSTRUCT}([\sqrt{v}]^2, H)$

Case 2. $|P| > 1$ and $v \leq v_0$ (we assume $v = v_0$): // *bitmap leaf*

1. compute the bitmap $M = \mathit{bitmap}(P)$ of size $v \times v$
2. store the concatenated rows of M as a vector W of length v^2 bits
   // *notice that* $v^2 = O(\sqrt{\log u})$ *and so* W *fits into one word*

Figure 1: The construct procedure builds the MDST data structure for approximate nearest neighbor queries.

The bitmap leaf is a leaf small enough to be processed directly by bitmap techniques that are stretched later. Let $v_0 = \lambda^{1/8}$ denote the minimal size of a block in the tree, where $\lambda = 1/\epsilon \cdot \log^2 u$. Rather than pursuing the recursive process until it ends with leaves of single points, we also stop the recursive process when $v \leq v_0$ and use a bitmap leaf to store that region. As shown later, this saves a $O(\log\log\log U)$ factor from the query complexity. In practice, however, it may be omitted for code simplification.

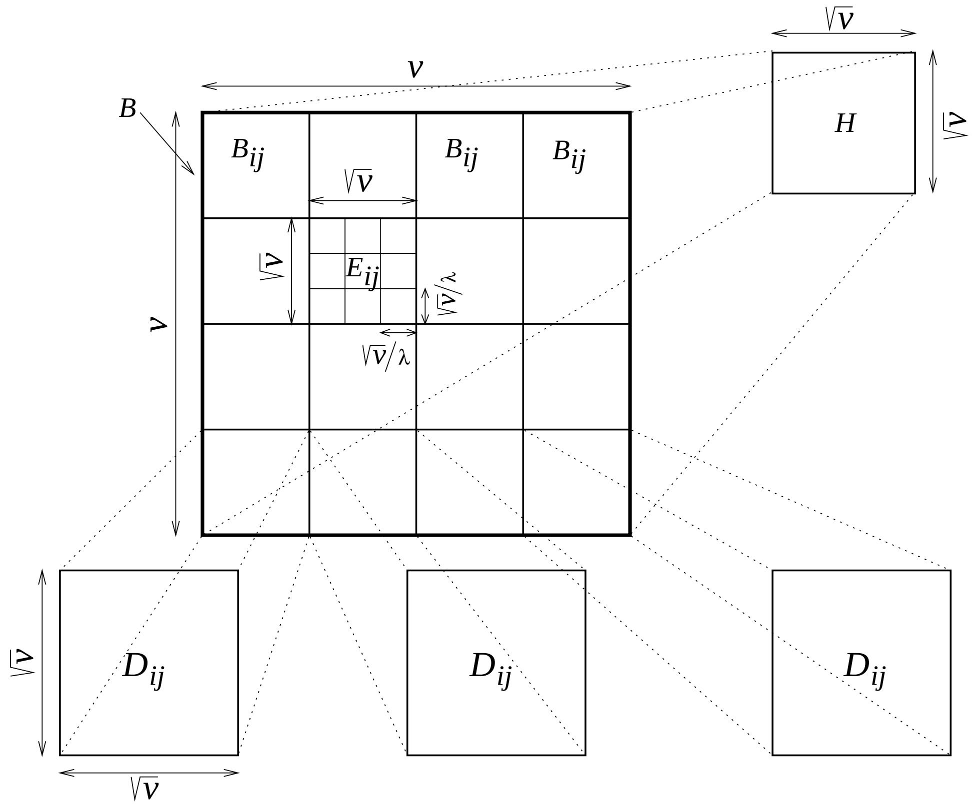


Figure 2: Illustration of the MDST data structure components at the internal node corresponding to a block B of size $v \times v$.

During the preprocessing we first randomly translate the data - this is why the root block is of size $[2U]^2$ instead of $[U]^2$. We also precompute certain lookup information used during the *search* procedure. The information consists of roughly $O(\log^{5/4} u)$ bits and can be computed in the same time. The precomputation procedure is as follows. Consider any point $q \in [-1/\epsilon \cdot v_0, 1/\epsilon \cdot v_0]^2$ and any integer r such that $B_r(q)$ (a disk of radius r centered at q), intersects but does not contain $[v_0]^2$. For each such a pair q, r we compute a binary matrix $M_r(q)$ of size $v_0 \times v_0$. The matrix has 1's at points p such that $d(p,q) \in [r, r+1)$ and 0's otherwise. Then we concatenate the rows of $M_r(q)$ forming a bit vector $A_r(q)$. Finally, we concatenate $A_r(q)$ for all r's into $A(q)$ and create a table mapping q to $A(q)$.

The procedure for finding an approximate nearest neighbor is described in Figure 1 and Figure 4. To avoid rounding details and to simplify the description we assume that both the input and output points are members of $[u]^2$ rather than $[v]^2$. Each type of node is handled separately. Again we focus first on the process in an internal node (case 1). In general there are three possible situations, illustrated in Figure 5. The algorithm first applies a (large) circle test, of radius $\sqrt{v}/\epsilon$, which in a sense divides the (unknown) bits of the result into two parts; the upper bits (most significant bits) and the lower bits (least significant bits). If there is no data point within the circle (Figure 5.a), then it is a lower bound on the distance to the nearest point which, within the approximation parameter ϵ, allows us to ignore the lower bits and continue the search for the upper bits only. That is, it is accurate enough to search for an actual point in H. Otherwise, we consider all the relevant blocks B_{ij} and use their E_{ij} coarse mapping to get a finer estimation of the distance to the closet point. If this is out of a (small) circle of radius $1/\epsilon\frac{\sqrt{v}}{\lambda}$ (Figure 5.b) then the actual point of the closest E_{ij} sub-block is the query result. Otherwise, there is a point in the small circle (Figure 5.c) and the algorithm proceeds recursively to the corresponding child D_{ij}. In such a case we already found the upper bits of the result. In the rare event where the small circle intersects two or more non-empty sub-blocks of different B_{ij} blocks, the algorithm has to recurse all non-empty intersected D_{ij} (up to four). This case is further explored in the complexity analysis.

This completes the description of the data structure and the algorithm. Now we proceed with the proofs. We first provide proof of correctness and then a complexity analysis.

2.1 Correctness

The correctness of the MDST search procedure will be proved in two steps. First, we prove that the data structures E_{ij} return approximate nearest neighbor, i.e. there is a constant c such that the returned point q is within the distance $(1 + c\epsilon)$ times the nearest neighbor distance. In the second step (and by using similar technique) we prove the correctness of the whole procedure.

The correctness of the data structures E_{ij} is shown as follows. First observe that none of the E_{ij}'s sub-blocks may contain any other E_{ij} structure (as $v = \lambda$ and therefore in the next level $\lambda > \sqrt{v}$). Therefore, each recursive call invokes either data structures D_{ij} (step 3.d) or H (case 2)). For simplicity we can assume that in step (3.d) the algorithm invokes only this D_{ij} which contains the closest point to p among all other data structures; since the actual algorithm invokes all D_{ij}'s, the above assumption does not influence the output. Due to this assumption we can represent the search procedure as a sequence of recursive calls of

SEARCH(P, q):

Case 0. $|P| = 1$: trivial. *// single-point leaf*

Case 1. $|P| > 1$ and $v > v_0$: *// internal node*

1. Let S be the set of (at most $1/\epsilon^2$) non-empty blocks B_{ij} within distance at most $\sqrt{v}/\epsilon$ from q.
2. if $S = \emptyset$, then

   return SEARCH(H, q)
3. otherwise $\quad // S \neq \emptyset$
   (a) let B' be the block in S closest to q and let r be its distance to q
   (b) let $S' = \{B_{ij} \in S : r \leq d(q, B_{ij}) \leq r + \sqrt{2}\sqrt{v}\}$
   (c) if E_{ij}'s exist then
      i. for each block $B_{ij} \in S'$, SEARCH(E_{ij}, q)
      ii. let r' be the distance to the closest *actual* point returned (say p)
      iii. if $r' \geq 1/\epsilon \frac{\sqrt{v}}{\lambda}$
            return p
         otherwise
            let R denote all (at most 4) blocks B_{ij} s.t. $d(q, B_{ij}) < r'$

      otherwise
         assign $R = S'$
   (d) for all $B_{ij} \in R$, SEARCH(D_{ij}, q) and output the closest point returned.

Case 2. $|P| > 1$ and $v = v_0$: *// bitmap leaf*

call procedure *bitmap*.

Figure 3: The search procedure for approximate nearest neighbor.

length 3; each call invokes either H or D_{ij}. Calling H might clearly result in an additive error of size bounded by the diameter of blocks B_{ij} (in the units of the universe $[u]^2$); however, the distance to nearest neighbor is at least $1/\epsilon$ times this quantity. On the other hand, calling D_{ij} involves *no error* at all. Let $e_1 \ldots e_k$ denote the additive errors incurred as above. As after each call of H the size of the region corresponding to a B_{ij} grows by a factor of at least 2, we can assume that the sum of e_i's is smaller than $2e_k$. On the other hand, we know that the distance to the actual nearest neighbor is at least $1/\epsilon \cdot e_k$. Therefore, the multiplicative error incurred is at most $(1 + 2\epsilon)$.

We can now proceed with the whole data structure. The recursive calls of the algorithm can be modelled in the similar way as above; however the algorithm has an additional option of stopping in step (3.c.iii). The

BITMAP(q):

1. if $q \notin [-1/\epsilon \cdot v_0,\ 1/\epsilon \cdot v_0]^2$ then output any point from P
2. otherwise
   (a) construct a word W' by concatenating $O(v_0)$ copies of W (by using one multiplication); notice that W' has length ${v_0}^3$
   (b) compute $A' = W'$ AND $A(q)$ (where $A(q)$ is a word prepared during the preprocessing)
   (c) find the first bit in A' set to 1 and return the point p corresponding to that bit

Figure 4: The Bitmap procedure (case 2, Figure 3)

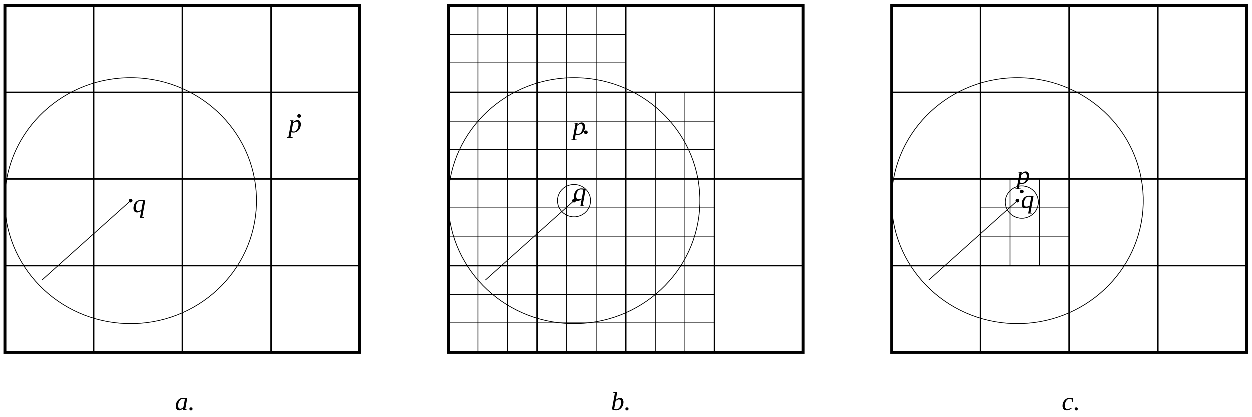


Figure 5: The three possible search situations in an MDST internal node.

latter case can incur an additive error bounded by the diameter of blocks C_{kl}; as the distance to the nearest neighbor is lower bounded by $1/\epsilon$ times the side of C_{kl}, the multiplicative error is at most $(1+\sqrt{2}\epsilon)$. It is easy to verify that the remaining cases are exactly as in case of E_{ij}.

In this way we proved the following lemma.

Lemma 2.1 *The distance from p to the point q returned by the algorithm is at most $(1 + O(\epsilon))$ times the distance from p to its nearest neighbor.*

2.2 Complexity

The complexity bounds for the procedures *construct*, *add* and *delete* are essentially as in [20]. Therefore, below we focus on the complexity of search, which follows from the following sequence of claims.

Claim 2.2 *Case 2 of the procedure* search *takes time* $C(\epsilon, u) = \lceil (1/\epsilon)^{3/8} / \log^{1/4} u \rceil$.

Proof: We show that the whole word W' fits within one word, then the procedure can be implemented in constant time. Otherwise (i.e. when $C(\epsilon, u) = \omega(1)$) we perform the same procedure sequentially on all words of W'.

The steps are implemented as follows. Step (a) is performed by multiplying W by a concatenation of v bit sequences, each consisting of a 1 followed by $v^2 - 1$ zeros. Step (b) uses just one boolean operation. The last step can be implemented using a constant number of boolean and arithmetic operations like in [10]. □

Claim 2.3 *For any* i, j *searching in* E_{ij} *takes time* $O((C(\epsilon, u) + 1/\epsilon) \cdot 1/\epsilon^3)$.

Proof: Recall that the data structure E_{ij} does not contain other E-type structures. Therefore, it contains only H-type and D-type structures. The recursive structure of E_{ij} can be represented as a tree. We observe that the depth of this tree is 3, as by using three recursive calls we reduce the block size from $v = \lambda$ (the size of E_{ij}) to $\lambda^{1/8} = v_0$. Searching in any "leaf" data structure (i.e. with $v = \lambda^{1/8}$) can be solved in time $C(\epsilon, u)$. The number such problems is at most $1/\epsilon^3$, as the set S' of data structures invoked recursively is of size at most $1/\epsilon$ and the level of recursion is 3. This incurs the first term of the cost function, i.e. $C(\epsilon, u)/\epsilon^3$. The second term follows from the fact that the complexity of step (1) is $1/\epsilon^2$ and this step is invoked at most $1/\epsilon \cdot 1/\epsilon$ times. □

Claim 2.4 *Consider step (3.d) of the search algorithm. If the input* P *is translated by a random vector, then:*

1. *the probability that* $|R| > 1$ *is at most* $O(\frac{1/\epsilon}{\lambda})$
2. *the probability that* $|R| > 2$ *is most* $O((\frac{1/\epsilon}{\lambda})^2)$

Proof: We might have $|R| > 1$ if p lies within distance r' of a boundary of some block B_{ij} (the chance is even smaller, as it also requires that the intersected E_{ij} is not empty) As the side of B_{ij} is $\sqrt{v}$, this event can happen with probability $O(\frac{r'}{\sqrt{v}}) = O(\frac{1/\epsilon}{\lambda})$. The other case involves p lying within distance $O(r')$ to a vertex of some B_{ij}; the probability of such event can be estimated in the same way. □

Claim 2.5 *If for all executions of step (3.d) the set R has cardinality at most 1, the procedure runs in $O((C(\epsilon, u) + 1/\epsilon) \cdot \log\log u \cdot 1/\epsilon^4)$.*

Proof: As each set R has cardinality 1, the recursion path is just a path of depth $\log\log u$. The cost of each step is dominated by the cost of invoking E_{ij} $|S'| = 1/\epsilon$ times. The cost estimate follows. □

Claim 2.6 *If for all executions of step (3.d) the set R has cardinality at most 2, the procedure runs in time $O((C(\epsilon, u) + 1/\epsilon) \cdot 2^{\log\log u} \cdot 1/\epsilon^4)$.*

Proof: The argument is similar to the above, but now two recursive calls are allowed, thus the size of the recursion tree is $2^{\log\log u}$. □

Claim 2.7 *The worst case running time of the procedure is $O((C(\epsilon, u) + 1/\epsilon) \cdot 4^{\log\log u} \cdot 1/\epsilon^4)$.*

Proof: As above, for the (very rare) case in which p lies within distance r' from a vertex of four non-empty B_{ij} blocks at all recursive levels along the search path. □

Lemma 2.8 *The expected running time of the* search *procedure is $O((C(\epsilon, u) + 1/\epsilon) \cdot \log\log u \cdot 1/\epsilon^4)$.*

Proof: Notice that

- if $|R| = 1$ for all executions of step (3.d), the number of times this step is executed is at most $O(\log\log u)$
- if $|R| \leq 2$ for all executions of step (3.d), the number of times this step is executed is at most $O(2^{\log\log u}) = O(\log u)$

The expected cost is then an order of

$$[1 \cdot \log\log u + (\log u \cdot \frac{1/\epsilon}{\lambda}) \log u + (\log^2 u \cdot (\frac{1/\epsilon}{\lambda})^2) \log^2 u](C(\epsilon, u) + 1/\epsilon) \cdot 1/\epsilon^4$$

which can be verified to be bounded by $O((C(\epsilon, u) + 1/\epsilon) \cdot \log\log u \cdot 1/\epsilon^4)$. □

2.3 Closest pair under insertions.

The closest pair maintenance under insertions can be reduced to dynamic approximate nearest neighbor (say with $\epsilon = 1$) as follows. First observe, that for any k we can retrieve k approximate nearest neighbors in time

$O(k \log\log u)$. To this end we retrieve one neighbor, temporarily delete it, retrieve the second one and so on, until k points are retrieved; at the end we add all deleted points back to the point-set. Next, observe that if we allow point insertions only, then the closest pair distance (call it D) can only decrease with time. The latter happens only if the distance of the new point p to its nearest neighbor is smaller than D. To check if this event indeed happened, we retrieve $k = O(1)$ approximate nearest neighbors of p and check their distance to p. If any of the point's distance is smaller than D, we update D.

To prove the correctness of this procedure it is sufficient to assume that the distance from p to its closest neighbor (say q) is less than D. Note that in this case there is at most a constant number of 1-nearest neighbors of q (as all such points have to lie within distance $2D$ of p but have pairwise distance at least D). Therefore one of the points retrieved will be the exact nearest neighbor of p, and thus D will be updated correctly.

3 Stratified trees for higher dimensions

In this section we present a multidimensional variant of stratified trees solving the approximate nearest neighbor problem in time $O(d + \log\log u + \log 1/\epsilon)$ under the assumption that $d \leq \log n$. The data structure is deterministic and static. We first describe a simple variant of the data structure which uses $d^{\log\log u} O(1/\epsilon)^d n^2 \log u$ storage; we then comment on how to reduce it to $d^{\log\log\log u} O(1/\epsilon)^d n \log u$.

The main component of the algorithm is a data structure which finds a d^2-approximate nearest neighbor in the l_∞ norm. Having a rough approximation of the nearest neighbor distance (call it R), we refine it by using the techniques of [11] to obtain a $(1+\epsilon)$-approximation in the following way. During the preprocessing, for any $r = 1, (1+\epsilon), (1+\epsilon)^2 \ldots$ (i.e. for $O((\log u)/\epsilon)$ different values of r) and for each database point p we build the following a data structure, that enables checking (approximately) for a query point q if q is within distance r from any database point which lies within l_∞ distance of $O(dr)$ from p (denote the set of such points by $N_r(p)$). The rough idea of the data structure is to impose a regular grid of side length $r/\sqrt{d}$ on the space surrounding q and store each grid cell within the distance of (approximately) r from $N_r(p)$ in the hash table (see [11] for details). The data structure uses $nO(1/\epsilon)^d$ storage for each r and p. The time needed to perform the query is essentially equal to the time needed to find the grid cell containing the query point q and compute the value of hash function applied to the sequence of all coordinates of that cell. In order to bound this time notice that after finding the d^2-approximate nearest neighbor of q we can represent (in time d) q's coordinates using $\log d/\epsilon$ bits per coordinate. Therefore, all coordinates of q can be represented using $O(d \log d/\epsilon)$ bits. Since we are allowed to perform arithmetic operations on words consisting of $d \leq \log u$ bits in constant time, it is easy to implement the hashing procedure using $O(\log d/\epsilon)$ operations.

In order to find a $(1+\epsilon)$-approximate nearest neighbor of q, we perform a binary search on $\log_{1+\epsilon} d$ values of r as described in [11]; this takes $O(\log(\log d)/\epsilon) \cdot O(\log d/\epsilon)$ operations, which negligible compared to $O(d)$.

Therefore, it is sufficient to find quickly a d^2-approximate neighbor of q. In order to do this, we apply a variant of multidimensional stratified trees described in previous section. Since the techniques are similar, we only give the sketch. The idea is to split the universe into squares of side $d\sqrt{u}$ (instead of $\sqrt{u}$), as long as $d^2 < \sqrt{u}$. Moreover, instead of using only one square grid as before, we use d of them, such that the ith grid is obtained from the first one by translating it by vector $(i\sqrt{u}, \ldots, i\sqrt{u})$. The reason for it is that for any point q there is at least one i such that the distance from q to the boundary of the cell it belongs to is at least $\sqrt{u}/2$; thus the correctness argument of the previous section goes through. Also notice that the depth of the data structure does not change (as in each step the universe size goes down by a factor of $\sqrt{u}/d > u^{1/4}$. However, the storage requirements are now multiplied by $d^{\log\log u}$, since at each level we multiply the storage by $O(d)$.

In order to bound the running time, we observe that during each recursive step the value of $\log u$ is reduced by a constant factor. Therefore, the description size of q (which is initially $d \log u$ bits long) is reduced by a constant factor (say c) as well, which means (by above arguments) that the i-step takes roughly $O(d/c^i)$ operations, as long as $d > c^i$. Thus, the total time is $O(d + \log\log u)$.

In order to reduce the storage overhead from $d^{\log\log u}$ to $d^{\log\log\log u}$, notice that the above analysis contains a slack - the time needed for the rough approximation is much larger than the time needed for the refinement. One can observe however that if during the refinement step each coordinate can be represented using $\log u/\log\log u$ bits, its running time is still $O(d)$. Therefore, we can stop the first phase as soon as log of the universe size drops below $\log u / \log\log u$, ie. after first $\log\log\log u$ steps.

The storage size dependence on n can be reduced to linear by using the covering technique of [11]. More specifically, we can merge those neighborhoods $N_r(p)$ which have very large overlap in such a way that that the total size of all neighborhoods is only linear and the diameter of merged neighborhoods gets multiplied by at most $O(\log n)$.

4 Quadtree Dilation in Linear Time

Given a shape S stored as a region quadtree, $T = T(S)$ as in Section 5.1, we present algorithm for computing the dilated region $D(S)$ in $O(n)$ time. The algorithm consists of two major parts. First it dilates blocks of certain size, denoted as *atom blocks* and then it merges the results in a DFS, bottom-up fashion. During the merging process, the algorithm computes and reports the vertices of $\partial D(S)$. Each of these parts consists of several steps which are briefly described below.

We use the following notations: let $R(T)$ denote the axis-parallel bounding square of the region occupied by T. For a node v of T, let T_v denote the subtree rooted at v, and let $R_v = R(T_v)$. Sometimes we refer to v (and its region R_v) as the *block* v. We say that v is a *grey block* if R_v contains both white (i.e. empty) and black (i.e. full) regions. Let d denote a direction, $d \in \{up, down, left, right\}$, let e_{down} denote the lower edge[5] of R, and let x_0 be a point on e_{down}. Let ℓ_{x_0} denote the vertical line passing through x_0. We define the *envelope point* of S with respect to R at x_0 in the down-direction, denoted $env(R, down)(x_0)$, as the lowest point of $\ell_{x_0} \cap R \cap S$, if $\ell_{x_0} \cap R \cap S \neq \emptyset$ and the distance of this point from e_{down} is at most r (otherwise $env(R, down)(x_0)$ is not defined). We define the (partially defined) function $env(R, down)(x)$, which is a polygonal x-monotone path(s) (see Figure 6). We define the *outer-path* of S in the down-direction, denoted by $op(R, down)$, as the collection of vertically-lowest points in $e_{down} \cup D(S \cap R)$ that lies below the line containing e_{down}. (see Figure 6). Observe that $op(R, down)$ is also a x-monotone path(s), consists of circular arcs of radius r, and of straight horizontal segments.

The next lemma, whose proof is easy and thus omitted, shows the importance of the envelope of a shape

[5] the same apply to all other three directions

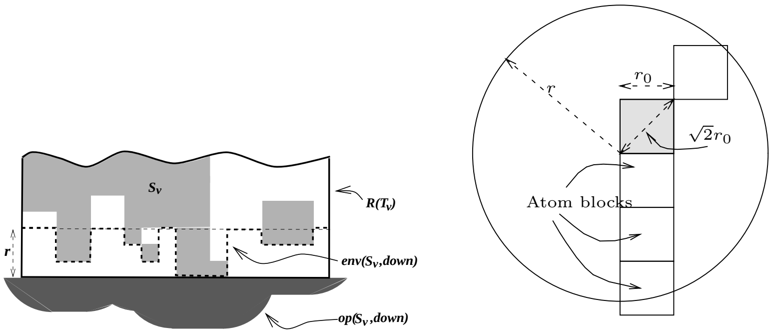


Figure 6: Left: the envelope $env(R_v, down)$ of the shape S_v is marked by the dashed line, and its dilation under the down-edge (hence its outer-path, $op(R_v, down)$) is filled in black. Right: the selection of r_0, given the dilation radius r.

term.

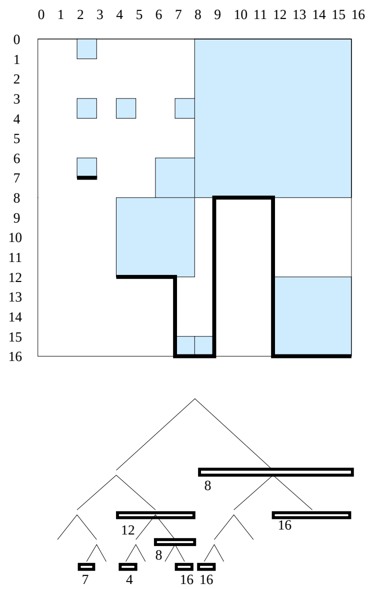
Lemma 4.1 *Let v and u be vertices of T, such that R_v and R_u are interior disjoint, then*

$$D(S_v) \cap R_u = D(\text{env}(R_v, d)) \cap R_u$$

where d is the direction at which R_u refers to R_v. That is, only $\text{env}(R_v, d)$ *counts in terms of influencing R_u by the dilation of S_v.*

Let $r_0 = 2^k$, for an integer k such that $\sqrt{2}r_0 \leq r < 2\sqrt{2}r_0$. We say that a block $v \in T$ is an *atom block* if the side of R_v is exactly r_0. Clearly, for a gray atom block v (which may contain as many as ${r_0}^2$ black and white blocks) $D(S_v)$ is a simply connected region that includes R_v; one can observe that r_0 is the side of the largest tree block having this property (see Figure 6 right). A *large block* (resp. *large leaf*) is any tree block (resp. leaf) larger in size than an atom block.

A crucial observation is that if q is a point in the plane, then there is only a constant number of atom blocks and large (black) leaves in the vicinity of q the dilation of which intersect q (no more than three atom blocks away at any given direction, or 32 blocks all around). We call the set of these blocks the *effective neighborhood* of v. Also, observe that all atom blocks and large leaves are interior disjoint, and their union covers $[u]^2$. The dilation algorithm first dilates each of these elementary regions by directly computing their outer paths, and then it computes the union of these dilated shapes in a bottom-up fashion. We will show



PROJECT(T_v):

1. Initialize an empty binary tree $T'_v = \emptyset$.
2. Traverse T_v in a DFS order and follow the projected path in T'_v.
   When visiting node $v_i \in T_v$:
   (a) if $(v'_i \notin T'_v)$
   insert(v'_i, T'_v)
   $y_{min}(v'_i) = \infty$
   (b) if $(color(v_i) = BLACK)$
   $y_{min}(v'_i) = min\{y_{min}(v_i), y_{min}(v'_i)\}$

Figure 7: Left: An atom block and its down-projection binary tree. Right: The projection algorithm.

how to compute $D(S_v)$ for an atom block v in a linear time, and then use these observations to compute the union in a linear time. These observations are the basis for the efficiency of our dilation algorithm.

Computing the Dilation of an Atom Block

The dilation of a gray atom block (of size $r_0 \times r_0$) is a simply-connected region. Therefore, it can be represented by one list of the arcs and straight lines along its boundary, which is a concatenation of the four outer paths of v at the four directions. By Lemma 4.1, the outer path at direction $d \in \{up, down, left, right\}$ can be computed from the envelope $env(R_v, d)$. Hence the dilation of the atom block requires the computation of its four envelopes and then its four outer paths.

To compute $env(R_v, down)$ we need to find the partition of $I_v = e_{down}(R_v)$ into intervals, to compute the y-location associated with each interval and to construct the envelope as a list. This is done in two steps. First we project T_v, the sub-quadtree rooted at v, into a binary tree, T'_v, and then we traverse the binary tree and compute the segments and the y-location of each segment. Let the node $w' \in T'_v$ denote the projection of a node $w \in T_v$. The path from v' to w' in T'_v is derived from the path from v to w in T_v by following the horizontal branches (and ignoring the vertical ones) along the path in t_v. An example is shown in Figure 4left.

The PROJECT(T_v) procedure traverses T_v in a DFS order, and simultaneously constructs and traverses T'_v. This is called a *projection*, as all the nodes $v_i \in T_v$ having (1) the same depth (in the quadtree), and (2) the same supporting x-interval, $I_{v_i} = I_{w'}$, are being projected to a single node $w' \in T'_v$, associated with this interval (e.g., the three small blocks which lie along one column in Figure 4 are projected to the same node (a leaf, in this example)). During the projection process, each node $w' \in T'_v$ maintains its $y_{min}(w')$ — the smallest y value of the down-edge of all the black leaves projected to it (if any. Otherwise $y_{min}(w') = \infty$).

The envelope is found in the second step by traversing T'_v in a DFS order. The partition of I_v into envelope segments is just the list of x-intervals associated with the leaves. The y location at an interval $I_{w'}$ is computed recursively during the DFS by finding the smallest y_{min} value among all the nodes along the path from the root v' down to the leaf w'. The time and space required for the construction of T'_v and its traversing is $O(n_v)$, where n_v is the size of T_v.

The next lemma shows that the result of this process is indeed the envelope. Let $path(T'_v, w') = \{v_1 = v', v_2, \ldots, v_k = w'\}$ denote the path in T'_v from the root v' to a node $w' \in T'_v$. Let $y_{min}(path(T'_v, w')) = min\{y_{min}(v'_i) \ : \ i = 1, \ldots, k\}$ denote the minimal value of $y_{min}(v'_i)$ that is encountered along $path(T'_v, w')$.

Lemma 4.2 *Let $x_0 \in u$, and assume $x_0 \in I_{w'}$ where w' is a leaf of T'_v. Then*

$$env(R_v, down)(x_0) = y_{min}(path(T'_v, w'))$$

Proof: Let $w_1 \in T_v$ be the black leaf (if any) in T_v which intersects the vertical line $x = x_0$ at the lowest y value, $y_0 = y_{min}(w_1)$. That is, $x_0 \in I_{w_1}$ and, by definition, $y_0 = env(R_v, down)(x_0)$. Let $w'_1 \in T'_v$ denote the projection of w_1. It follows that $I_{w'} \subseteq I_{w'_1}$ (w' is a leaf in T'_v), and $w'_1 \in path(T'_v, w')$. Hence $y_{min}(path(T'_v, w')) \leq y_{min}(w'_1) \leq y_{min}(w_1) = y_0$. The selection of w_1 ensures that there is no lower black leaf on the path, that is, $y_{min}(path(T'_v, w')) \geq y_{min}(w_1) = y_0$. If there is no such a black leaf $w_1 \in T_v$, then $y_{min}(v'_i) = \infty \ \ \forall \ v'_i \in path(T'_v, w')$. □

Computing the Outer-Path of an x-monotone path We will describe the algorithm for one direction, $d = down$. Let ℓ be a horizontal line, and let C be an x-monotone piecewise-linear path, lying completely above ℓ. (In our application, l is the down-edge of a block and C is the down envelope). We need to compute

$op(C, down)$, the region of $D(C)$ which is below ℓ (see Figure 6). To perform this task we need the following lemma, taken from [6];

Lemma 4.3 *Assume C consist of C_l, a left part, and C_r, its right part, where C_l is completely to the left of C_r. Then $op(C_l, down)$ and $op(C_r, down)$ intersect at most once.*

We scan C from right to left, and process one segment e of C at a time. Each such segment is a constant-y piece. Let α, α' denote the right and left points of the current segment, e, respectively. Let C_α denote the part of C which is to the right of α. Let $opd(\alpha)$ denote $op(env(C_\alpha, down))$. Assume that we have already computed $opd(\alpha)$, and let β be the leftmost point of $opd(\alpha)$. We seek $opd(\alpha')$. For this, we only need to find the intersection point q (if it exists) of $opd(\alpha)$ and the region of $D(e)$ below ℓ. Next, we remove the part of $opd(\alpha)$ lying between q and β, and concatenate the "new" region of $opd(\alpha')$ which lies between q and the leftmost point of $op(e, R, down)$ (the outer path of e).

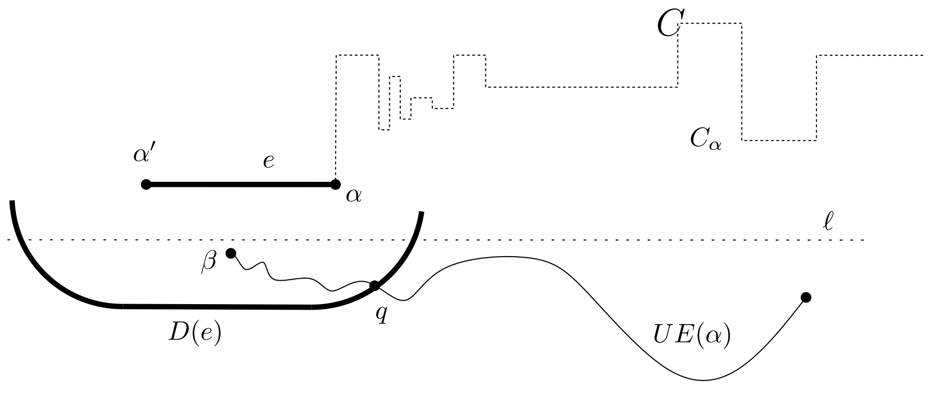


Figure 8: Computing the outer path

Finding and deleting the part $q\beta$ from $opd(\alpha)$ is achieved as follows: We traverse $opd(\alpha)$ *right*, starting from β, as long as we are in $D(e)$. q is the point at which we leave $D(e)$. Lemma 4.3 ensures that no other intersection point exists. The time needed for computing $opd(\alpha')$ (after $opd(\alpha)$ has been determined) is proportional to the complexity of the deleted portion $q\beta$. Since each part in the region is created only once, and can be removed only once, and since the number of elements in $opd(\alpha)$ is proportional to the complexity of C (by [13] sited above) the execution time over the course of the procedure is linear in the complexity of C.

The Merging (Zipping) Process

To explain the dilation merging procedure we need the following definition: For any large block v, let $s_{v,d}^{(i)}$ denote the i-th *corridor* of R_v at direction d, for $i = 0, 1, 2$. This is a maximal length, r_0-wide rectangle contained in R_v, that lie along the d-edge of R_v at a distance ir_0 from that edge (see Figure 9). A corridor is represented by a double-linked list of all the atom blocks it contains and the large leaves it intersects, in their appearance order along the corridor. Corridors play a central part in our algorithms, and are called *the corridors associated with* v. Each block v which is an atom block or larger maintains its corridors and their envelopes.

The dilation merging process (zippering) takes place in large gray blocks. First, we construct the data structure associated with each block v, using the data structures of its children. Then it processes the *active zipper area* — those parts of its child's corridors which are not included in its own corridors. These corridor parts are found near the edges shared by two children (see Figure 9). It is easy to see that this region, denoted as the *active* (or the zipper) region in the figure, is disjoint from the dilated area of any part of S which is out of R_v. Moreover, all the effective neighborhood of this area is either in corridors or in interior regions (for which $D(S)$ has already been computed).

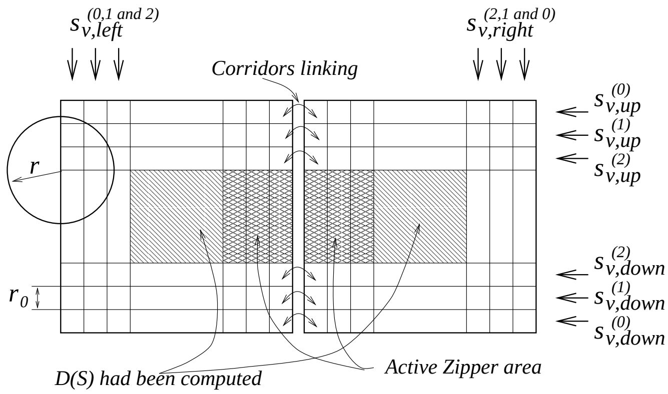


Figure 9: Two large blocks are shown, during their merging (zipping) process.

The main invariant of the algorithm is the following: when the algorithm exits a vertex v towards its parent, all vertices of $D(S_v)$ whose distance to the boundary of R_v is at least $3r_0$ have already been reported (see Figure 9). More precisely, before leaving v we compute the vertices of $D(S_v)$ which lie in those parts of the corridors of its four children that do not intersect with its own corridors (note that v's corridors are a subset of the union of its children' corridors). It also updates relevant data structures required for further steps of the algorithm. This process is called *merging* or *zipping* of blocks.

Next we explain how to compute all the intersections between two outer paths φ and ψ. We later extract from the results the vertices of $D(S)$.

If φ and ψ are both outer-paths in opposite directions, (e.g. φ is defined downward and ψ is define upward) then the problem is solvable in linear time using a standard merge-like procedure. Therefore we assume that this is not the case. Assume for example that φ is defined on a horizontal direction, so let $\varphi = op(R_1, up)$ and $\psi = op(R_2, right)$. See Figure 10.

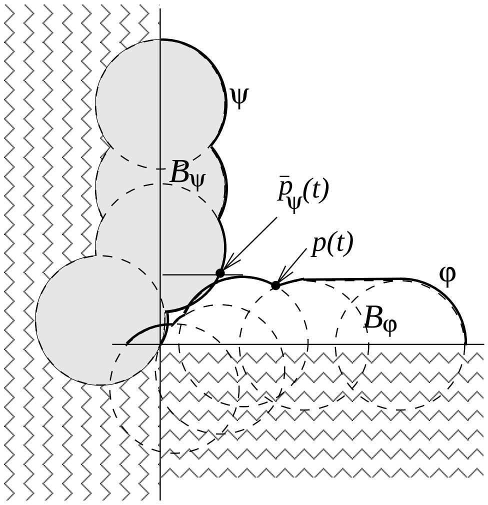


Figure 10: *The merge process between a right outer path ψ (coming from the vertical zigzag region) and an up outer path, φ (coming from the horizontal zigzag region). The point p "traverses" along φ, and at the time of this snapshot all the three intersection points between the two outer paths have already been reported. Note, however, for the position of $\bar{p_\psi}$, which keeps the highest projection point of p on ψ seen so far.*

Let B_ψ (resp. B_φ) denote the union of disks and rectangles contributing parts to ψ. We traverse φ, from left to right, starting from the leftmost point to the right of the right edge of ψ (the horizon of ψ). Let $p(t)$ (for $t > 0$) denote the location at time t. Let $p_\psi(t)$ (see Figure 10) denote the point which is the horizontal projection of $p(t)$ on ψ, and let $\bar{p}_\psi(t)$ denote the highest point in

$$\{p_\psi(t') \,|\, t' \leq t, \text{ and } p_\psi(t') \notin B_\psi\} \ .$$

The reason for this odd-looking definition is that it is possible (and sufficient) to maintain $\bar{p}_\psi(t)$, while we are not able to maintain the position of $p_\psi(t)$ efficiently, since if a point $p(t) \in \varphi$ is outside $B_\psi(t)$, than so are all points of φ which is to the right of $p(t)$, and not higher than $p(t)$. We maintain $\bar{p}_\psi(t)$ as follows. At $t = 0,\ p(0)$ is either inside B_ψ, or to the right of ψ. We distinguish between these two situations by scanning the arcs of ψ from bottom upward and seeking for an arc whose y-span contains the y-coordinates of $p(0)$. The algorithm now iterates between the following two modes:

First mode: $p(t) \in B_\psi$. As long as $p(t)$ is outside B_ψ, $p(t)$ traverse φ to the right, while maintaining the position of $\bar{p}_\psi(t)$ on ψ. This is doable using a merge-like process, in amortized time $O(1)$ per arc of φ. Once $p(t)$ and $\bar{p}_\psi(t)$ coincide, we are at a new intersection point v_1 of ψ and φ. We record this point, and switch to the second mode.

Second mode: $p(t) \notin B_\psi$. Assume that $p(t)$ is at the point v_1 that lies on the arc γ_φ of φ and the arc γ_ψ of ψ, which in turn lies on a disk or rectangle b_ψ of ψ. We check whether the endpoint u of γ_φ to the right of v_1 is inside b_ψ. If yes, we set γ_ψ to be its successive upper arc of ψ, and repeat checking (that is, traversing upward along ψ), until the u is outside b_ψ. Then γ_φ has intersection point v_2 with the boundary of b_ψ (— but not necessarily with γ_ψ). If v_2 also lies on γ_ψ (and hence on ψ) then v_2 is another intersection point of ψ and φ by which we "leave" B_ψ; We set $\bar{p}_\psi(t)$ to v_2, and switch to the first mode. In the second case (i.e. v_2 does not lie on ψ) we conclude that v_2 is inside another disk or rectangle b' of B_ψ, that contributes an arc γ'_ψ to ψ. A key observation is that γ'_ψ is above γ_ψ, since all centers of disks of B_ψ are above all centers of disks of φ. Thus we proceed advancing on ψ, until we found the first disk or rectangle $b'_\psi(t)$ that contains v_2 (b' must exist, otherwise v_2 were a point of ψ). We repeat the whole process of the second mode at this stage. Since in each stage we proceed either in φ or on ψ, the time needed for the whole process is linear in the complexity of φ and ψ.

5 Quadtree Complexity, Construction and Point-location

5.1 Complexity of Segment Quadtree

Fat objects. A planar convex object c is α-*fat* if the ratio between the radii of the disk s^- and s^+ is at least α, where s^+ is the smallest disk containing c and s^- is a largest disk that is contained in c (for fat, non-convex objects, the definition is more involved. Here we consider only convex fat objects). Let $\mathcal{C}$ be a collection of n convex α-fat objects in the plane, for a constant α, such that the combinatorial complexity of each of them is a constant, and the boundaries of each pair of objects intersect $\leq s$ times for a constant s. Let $\lambda_s(n)$ denote the maximal length of a (n, s)-Davenport-Schinzel sequences (see [19]). It is known that $\lambda_s(n)$ is almost linear in n for any constant s. It is shown in [7], (see also [9]), that the number N of vertices of $\partial \cup \mathcal{C}$ is only $O(\lambda_{s+2}(n) \log^2 n \log\log n)$, for an appropriate constant s.

Theorem 5.1 *Let T be a segment quadtree, constructed for the union of $\mathcal{C}$. Then the number of leaves in T is $O(N \log u)$, provided that κ, the maximal number of arcs and vertices of $\partial(\cup\mathcal{C})$ stored at each leaf of T is a large enough constant, that depends on α.*

Proof: Set $\mathcal{L}$ to be a level of T (that is, a collection of all nodes of T whose distance from the root is the same). We will show that the number of leaves of T in $\mathcal{L}$ is $O(N)$. The proof of the theorem is obtained by summing this bound over all $\log_2 u$ levels of T.

Let v be a leaf node of $\mathcal{L}$. The complexity of $R_{parent(v)} \cap \partial(\cup\mathcal{C})$ is larger than κ (otherwise $parent(v)$ would have been a leaf). Let $\bar{R}$ be a square whose center coincides with the center of R_v, and whose edge-length is larger by some factor $\kappa' > 1$, than the edge-length of R, where $\kappa' > 1$ depends on α. Using arguments similar to the ones used in [8], we can show that at least one of the following cases must occur:

- There is an object $c \in \mathcal{C}$ such that $\bar{R}$ contains z, a rightmost, a leftmost, a highest or a lowest point of ∂c.
- $\bar{R}$ contains a vertex z of $\partial(\cup\mathcal{C})$.
- There is an object $c \in \mathcal{C}$ such that the area of $\bar{R} \cap c$ is at least a constant factor of the area of c.

In the first two cases z can be charged by at most κ' blocks v, and the same holds for the region of c that we charge in the third case. Since the number of vertices z of $\partial(\cup\mathcal{C})$ is $O(N)$ and the number of elements that we charge in the first and third case is $O(n) < O(N)$, we conclude that the number of leaf-nodes v in $\mathcal{L}$ is $O(N)$, as asserted. $\square$

Let $T = T(S)$ denote a (region) quadtree representing a shape S, and let N_l denote the number of leaves in T. It is known that since S consists of $\leq N_l$ disjoint convex objects, namely the black blocks, then $\partial D(S)$ contains $O(N_l)$ vertices. See [13]. Combining this bound with the fact that the dilation of a black block of T is always a fat region, we can show the following lemma, using exactly the same argumentation as in the proof of Theorem 5.1.

Lemma 5.2 *Let $\tilde{T}$ be a segment quadtree constructed for $\partial D(S)$. Then the number of leaves in $\tilde{T}$ is $O(N_l \log u)$.*

Remark Recall that the number of nodes in a compressed (resp. uncompressed) quadtree is at most two (resp. $\log u$) times the number of leaves.

5.2 Constructing Quadtrees

Storing D(S). The above discussion shows that $D(S)$ can be efficiently stored in a quadtree. In Section 4 we show how $D(S)$ can be computed in (optimal) linear time. We later employ this algorithm in order to store the result in a convenient form, e.g. a segment quadtree (this requires only a slightly higher complexity).

In this section we show that it is convenient to store the output of the dilation algorithm of Section 4 as a segment quadtree.

Theorem 5.3 *Let S be a planar shape given as a region quadtree T, consisting of n nodes, and let r be a given radius. We can construct a segment quadtree $\tilde{T}$ that stores $D(S)$ in time and space $O(n \log^2 u)$.*

Proof: The constructive proof requires the following definitions: A *balanced* quad tree is a quadtree, with the additional property that if v_1 and v_2 are nodes in the tree such that R_{v_1} and R_{v_2} shares an edge or a portion of an edge, then the depth of v_1 differs by at most 1 from the depth of v_2. A quadtree of size n can be balanced by adding $O(n)$ additional nodes ([3, Theorem 14.4]). A *netted* quadtree is a quadtree at which, if R_{v_1} and R_{v_2} are neighboring squares, then there exists a pointer, called a *net pointer* from v_1 to v_2 and from v_2 to v_1 [17]. The combination of both attributes guarantees that only a constant number of net pointers are attached to each node.

We can now describe the algorithm. We start with an empty output tree $\tilde{T}$. We maintain $\tilde{T}$ both netted and balanced, and all its nodes are stored in a hash table H. We run the dilation algorithm of Section 4, whose output can be arranged (that is, directly reported) as a collection of closed curves of $\partial D(S)$. The output quadtree is being built in two phases.

First, we perform the following procedure for each of the closed curves. Let C be such a curve. We pick an arbitrary point q of C, and find the leaf-cell v of $\tilde{T}$ containing q, using H, by the point-location data-structure of Section 5.3. We insert the arcs of C into v, by the order they appear along C. If at some point the number of arcs in v exceeds the threshold κ of $\tilde{T}$, we split v, and accordingly update the hash table, the net pointers, and perhaps split node(s) in order to keep the tree balanced (which may require further splits, but, as quoted above, it sums up to a linear number of additional nodes). If, on the other hand, the arc γ of C we follow intersects with the boundary of R_v and 'leaves' this block, then we use the net pointers to find the neighbor leaf to which γ 'enters', and continue the process in that block. As noted, each node has only a constant number of neighbors to keep track of.

Next, we have to scan the empty block (that does not intersect $\partial D(S)$) and label each such block whether it is fully inside or outside $D(S)$. This is done using an algorithm similar to connected-component-labeling algorithm [17]: We traverse the tree, while adding all the non-empty leaves to a queue. Then, it pops one leaf at a time, uses it to label its (yet unlabeled) empty neighbors, and adds only those newly-labeled nodes to the queue.

When bounding the running time of this algorithm, it is clear that phase two takes only a linear time. For phase one, we first have to calculate the time needed to perform all the single point-location operations, one per boundary path of $D(S)$. Their number cannot exceed $O(n)$ (and is assuringly much smaller). Using the technique from Theorem 5.5, each one takes $O(\log \log u)$ time, or a total of $O(n \log \log u)$. The remaining running time is proportional (since no vertex is ever deleted) to the number of nodes created in $\tilde{T}$, which we denote by m. By Lemma 5.2 we know that $m = O(n \log^2 u)$, we thus have proven Theorem 5.3. $\square$

The size of the resulting quadtree can be improved by a factor of $\log u / \log\log n$ by using compressed quadtree (as in this case the size is proportional to the number of leaves). The following lemma shows that in this case the running time is improved by almost the same factor.

Lemma 5.4 *Given n* interior disjoint *objects, a compressed quadtree representatation of their union can be computed in time $O(N \log\log u)$, where N is the size of the resulted quadtree.*

Proof: *The idea is similar to the algorithm described above. The difference is that we need now to perform the point location in $O(\log\log u)$ time in a compressed tree, instead of $O(\log u)$ as before. Also, the time needed to jump between neighboring leaves is no longer constant, as we cannot afford to maintain balanced tree; thus, we will have to make sure that this operation has to be done in $O(\log\log u)$ time as well. We use the corresponding technique from Theorem 5.5, i.e. hash $O(\log\log u)$ paths per tree edge to achieve $O(\log\log u)$ query time. The total time spent on updating the hash tables is $O(N \log\log u)$, which is within the bounds.*

The last point we have to verify is that while constructing the output quadtree we do not generate long paths which should be replaced by single edges in the compressed quadtree; this can be easily done by applying unbounded binary search.

$\square$

By applying the technique from the above Lemma to Theorem 5.3 we obtain an algorithm for computing a compressed quadtree of dilated shape in $O(N' \log\log u)$ time, where N' is the size of the quadtree.

5.3 A point-location data structure

Theorem 5.5 *Let S be a planar shape consisting of union of cells of the integer grid $[u]^2$, and given as an uncompressed quadtree T, consisting of n nodes. Then in time $O(n)$ we can construct a data structure of size $O(n)$, such that given an integer query point q, we can determine whether q lies inside S in (expected) time $O(\log\log u)$. If T is a compressed quadtree, then we can construct the data structure in time and space $O(n \log\log u)$ and expected query time $O(\log\log u)$.*

Proof: We store the nodes of T in a hash table. When T is an uncompressed quadtree, the key of each node v is the binary representation of the path from the root of T to v, which is merely the position in $[u]^2$ of the block v. Using a binary search on the height of the tree, checking for each level probed, if there is a node of T at that level which lies on the path leading to q, we can find a block containing q, or determine that no such block exists, in expected time $O(\log \log u)$. This idea has appeared in the literature [21].

When T is a compressed tree, the analysis is a bit more complicated. This is due to the fact that simple replication of the previous approach would seemingly require hashing all the paths in the tree, which would imply $O(nt)$ storage requirement, where $t = O(\log u)$ is the depth of the tree. However the following observation allows us to reduce the storage to $O(n \log \log u)$. Call an interval $\{i \ldots j\} \subset [t]$ a *primitive* interval if $i = k2^p$ and $j = (k+1)2^p$ for some k and p. Each such interval corresponds to a subtree of a binary tree decomposition of $[t]$. Thus $[t]$ can be decomposed into primitive intervals in a tree-like fashion, by finding the largest primitive interval J in $[t]$, removing J from $[t]$ and recursing on $[t] \setminus J$. One can observe that such a decomposition is unique, and $[t]$ is decomposed into $O(\log_2 t)$ primitive intervals, since for every p, at most one primitive interval of length 2^p might appear in the decomposition.

The algorithm proceeds as follows. Each compressed edge is split into $O(\log t)$ primitive intervals. Instead of hashing all paths, the algorithm hashes only the paths with endpoints at primitive intervals. The following claim, whose proof is trivial and omitted, shows that this restriction does not influence the behavior of binary search procedure, and thus concludes the proof of Theorem 5.5. ◻

Claim 5.6 *Let e be a compressed edge in the quadtree from level i to level $j > i$ and let l be any level probed during the binary search for any point q. Then l has to be an endpoint of one of $O(\log(j-i+1))$ primitive intervals from the decomposition of the interval $[i \ldots j]$.*

6 Discussion

This paper provides optimal and near optimal algorithms for several problems in computational geometry on the integer grid. We show that under the assumption of that the input points have limited precision (i.e., are drawn from the integer grid of size u) these data structures yield efficient solutions to many important problems. In many cases we are able to replace $O(\log n)$ with $O(\log \log u)$. This requires an efficient evaluation of the $\log u$ bits required to write the result. We solve this problem using regular data structures. Some of these regular data structures, like the quadtree, can be stored in hash tables, which allow us to process, for example, a point location query without having to traverse the whole path from the root to the leaf (which would take $O(\log u)$). In other cases, like with the MDST, the regular data structure divides the location bits at each recursive level into two significantly large parts, and therefore the height of the tree becomes $\log \log u$. Most of these algorithmic problems have otherwise $\Omega(\log n)$ lower bounds in standard algebraic tree model, assuming arbitrary precision input. These basic ideas can be used to improve the efficiency of many other algorithms.

In several of the algorithms we stop the recursive process before it reaches the trivial type of leaf, and handle the rather complex "leaf" using a separate, non recursive algorithm. For example, in the quadtree dilation algorithm, if we try to eliminate the notion of an atom block, and just apply the dilation and merge right from the quadtree leaves, the complexity will no longer be linear and independent of R. In order to achieve the desired efficiency we must identify the right level at which the recursive process must be stopped and replaced with a non recursive algorithm.

In some situations the random translation in the MDST algorithm and the following expected complexity bounds are undesired. In such a case, the problem that occurs near the borders between B_{ij} blocks can be transferred from the query time to the update time. If one extends the borders of the B_{ij} blocks to have them overlap in a $2r'$-wide region, then there will always be a block that contains the entire small circle around the query point q (see Figure 5.c). Therefore, there is no need to recourse into more than one child (i.e., $|R| = 1$ at all times). This requires, however, some modification of the insert and delete procedures, which in turn should allow insertion/deletion of a point into/from all the blocks it intersect with.

References

[1] Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R., and Wu, A. Y. An optimal algorithm for approximate nearest neighbor searching. In *Proc. of the Fifth Annual ACM-SIAM Symp. on Discrete Algorthms* (1994), 573–582.

[2] Beame, and Fich. Optimal bounds for the predecessor problem. In *STOC: ACM Symposium on Theory of Computing (STOC)* 1999, 295–304.

[3] de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Berlin, 1997.

[4] de Berg, M., van Kreveld, M., and Snoeyink, J. Two- and three-dimensional point location in rectangular subdivisions. *J. Algorithms 18* (1995), 256–277.

[5] Dillencourt, M. B., Samet, H., and Tamminen, M. A general approach to connected–component labeling for arbitrary image representations. *Journal of the ACM 39*, 2 (1992), 253–280.

[6] Efrat, A., and Itai, A. Improvements on bottleneck matching and related problems using geometry, *Proc. 13th Annu. ACM Sympos. Comput. Geom.* (1996), 1996, 301–310.

[7] Efrat, A. The complexity of the union of (α, β)-covered objects. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, 1999, 134–142.

[8] Efrat, A., Katz, M. J., Nielsen, F., and Sharir, M. Dynamic data structures for fat objects and their applications. In *Proceedings of the 5th Workshop on Algorithms and Data Structures (WADS)*, 1997, 297–396.

[9] Efrat, A., and Sharir, M. On the complexity of the union of fat objects in the pl ane. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, 1997, 104–112.

[10] Fredman, M. L., and Willard, D. E. BLASTING through the information theoretic barrier with FUSION TREES. In *Proceedings of the Twenty Second Annual ACM Symposium on Theory of Computing* (1990), 1–7.

[11] Indyk, P., and Motwani, R. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *STOC: ACM Symposium on Theory of Computing (STOC)*, 1998, 604–613.

[12] Johnson, D. B. A priority queue in which initialization and queue operations take $O(\log(\log(D))$ time. *Mathematical Systems Theory 15* (1982), 295–309.

[13] Kedem, K., Livne, R., Pach, J., and Sharir, M. On the union of Jordan regions and collision-free translational motion amidst polygonal obstacles. *Discrete Computational Geometry 1* (1986), 59–71.

[14] Muller, H. Rastered point locations. In *Proceedings of Workshop on Graphtheoretic Concepts in Computer Science*, 1985, 281–293.

[15] Overmars, M. H. Range searching on a grid. *Journal of Algorithms 9* (1988), 254–275.

[16] R. G. Karlsson, J. I. M. Proximity on a grid. In *Proceedings of the Second Symposium on Theoretical Aspects of Computer Science*, 1985, 187–196.

[17] Samet, H. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[18] Samet, H. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[19] Sharir, M., and Agarwal, P. K. *Davenport-Schinzel Sequences and Their Geometric Applications.* Cambridge University Press, New York, 1995.

[20] van Emde Boas, P. Preserving order in a forest in less than logarithmic time. *Information Procesing Letters 6* (1977), 80–82.

[21] Willard, D. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Information Procesing Letters 17* (1983), 81–84.