Efficient Regular Data Structures and Algorithms for 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 [24]. 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

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., [22, 23]), which is based on a recursive decomposition of a square into four quadrants; another example is the stratified tree structure by van Emde Boas [24]. 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 [7, 23])

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[1] objects. It is therefore important to study such cases, for both theoretical and practical 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 in-

*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 NSF grant CCR-9623851

‡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-92-16970 and the Department of Energy under Contract DEFG0295ER25237.

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

put coordinates are integers from the interval $[u] = \{0 \ldots u-1\}$ 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.

Our first sequence of results apply to problems on sets of *points*. For this case, we use multidimensional generalizations of stratified trees by van Emde Boas [24]. The idea will be described, for clarity, for the two-dimensional case, but extends to higher dimensions. It starts by splitting $[u]^2$ into u equal squares of side $\sqrt{u}$. The non-empty squares are stored in a hash array, so that we can verify if a given square is non-empty in constant time. Then the construction is applied recursively inside each square of sufficiently large (though still constant) complexity. Finally, an array A of the size $[\sqrt{u}]^2$ of points is formed where each black point corresponds to a non-empty square; the construction is then applied recursively on A.

The 1-dimensional version of this construction cited above yields a dynamic data structure for nearest-neighbor queries. The running time which it guarantees (i.e. $O(\log\log u)$) has been recently improved to $O(\log\log u/\log\log\log u)$ [5]. We are not aware, however, of any prior work in which its multidimensional version has been applied.

The first data structure allows us to achieve $O(\log\log u)$ time per operation for dynamic approximate nearest neighbor (when insertions and deletions of points are allowed) and exact on-line closest pair (when insertions are allowed). Both results hold for any fixed dimension (the dependence on d is exponential). The data structure is randomized and the bounds hold in the expected sense.

The second 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 $d^{\log\log\log u} O(1/\epsilon)^d n \log^{O(1)} u$-space. Recently, Beame and Fich [5] showed a lower bound of $\log\log u/\log\log\log u$ 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 $d = O(\log n)$.

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 (occasionally we use stratified trees for better running times).

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

Before describing the results in more details, we need the following definitions: The *size* of a quadtree is the number of nodes in the tree. Among the many variants of 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)

segment (or *mixed*) quadtree: we allow the leafs to contain shapes of constant complexity $\leq \kappa$ (e.g. at most κ points).

compressed quadtree: a variant of either region quadtree or mixed queadtree, 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; note that the size of the resulting tree is at most twice the number of its (non-empty) leaves.

Firstly, we address the issue of efficiency of a quadtree as a planer shape representation. As mentioned earlier, in the worst case, the size N of the quadtree could be much larger than the complexity of the shape it represents. However, we show that if the shape is consists only of collection of fat convex objects (formally, is a union of n fat objects in $[u]^2$), then it can be represented as a segment quadtree with $N = O(M \log u)$ leaves where M is the complexity of the union (which is known to be almost linear [9]). This gives $O(M \log u)$ bound for the *size* of compressed quadtree and $O(M \log^2 u)$ for the size of uncompressed one. The quadtree can be efficiently constructed given the decomposition of the shape into a union of n *disjoint* objects (not necessarily fat) Given such a union we can compute the quadtree in time $O(N \log\log u)$, where N is the size of the quadtree.

Next, we address the efficiency of operations on a quadtree. It is easy to see that the tree has depth $O(\log u)$ and thus point location can be performed within this bound. However, by performing binary search on levels of the tree, one can reduce this time to $O(\log\log u)$, with preprocessing time and space linear in N [25]. We show that the same query time holds for *compressed* quadtree, with time/space multiplied only by a factor of $O(\log\log u)$.

We show that given a region quadtree of size N and a number $r > 0$, the *dilation* (Minkowski sum) of the shape which is represented by the quadtree with a ball of radius r can be computed in optimal time $O(N)$. Both point location and dilation are common and important operation in 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.

Most of the algorithmic problems above have $\Omega(\log n)$ or $\Omega(n \log n)$ lower bounds in standard algebraic tree 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 the running time bound. Notice that in most situations this change yields an 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 [20], the point location problem [18] and orthogonal range searching [19]; the solutions given in these paper give static data structures for two-dimensional data with query time $O(\log\log u)$. A number of offline problems has been also considered (see [19] 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 [19, page 273].

2 Dynamic multidimensional stratified trees

In this section we present a dynamic data structure for the approximate nearest neighbor problem in $[u]^d$. Let $P \subseteq [u]^d$ be a set of points, and let $\varepsilon > 0$ be a predetermined parameter. Given a query point $q \in [u]^d$, the data structure enables us to find an approximated nearest neighbor $p_{\text{app}} \in P$ to q; that is, a point such that $d(q, p_{\text{app}}) \leq (1+\varepsilon) d(q, p_{\text{nn}})$, where $p_{\text{nn}} \in P$ is the (exact) closest neighbor to q (see, e.g. Arya et al, [4]). The query time and the update time (inserting or deleting a point) is $O(1/\epsilon^{O(d)} \log\log u)$ in $[u]^d$ (for $d \geq 2$). By an easy reduction we also show how to maintain *exact* closest pair under insertions of new points in the same time. We describe, for clarity, the algorithm in two dimensions, but it can be easily extended to higher dimensions.

We resort to a different space subdivision technique called *stratified trees* proposed by **van Emde Boas** [24]. The structure supports nearest neighbor query in the 1-dimensional integer interval $[u]$ in $O(\log\log u)$ time. The structure is dynamic and each addition or deletion of a point takes $O(\log\log u)$ time. Let n denote the maximum number of points added to the data structure, then $O(n)$ space is used by the data structure[3]. It is assumed that one can perform standard boolean and arithmetic operations on words of size $\log u$ in constant time; some of our results require this assumption. Here we present a multidimensional extension of that data structure. To our knowledge no extension of this data structure to multidimensional spaces has been previously proposed.

The data structure 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 find an approximate nearest neighbor of a given point. Below we give the description of *construct* (together with the description of the data structure per se) and *search*. The (nontrivial) implementation of the *add* and *delete* are essentially the same as in the one dimensional case, therefore the reader is referred to [16] for details.

The first procedure, *construct*, takes as an input a set of points P from the universe $[v]^2$. As the procedure is used recursively, v does not have to be equal to u (but is its divisor). Therefore, the universe $[v]^2$ does not in general coincide with $[u]^2$; rather than that, it contains *sub-squares* of $[u]^2$ of side u/v. This also means that the set $P \subset [v]^2$ is not in general a subset of the initial data set. Notice that for any $p \in P \subset [v]^2$ we can quickly retrieve (after some preprocessing) *some* actual data set point belonging to the sub-square represented by p; we refer to such a point as an *actual* point of p.

The construct procedure is described in Figure 1. Before we apply the procedure, we shift the set of points by a random vector from $[v]^2$; this expands the universe from $[v]^2$ to $[2v]^2$, therefore we assign $2v$ to v before running the procedure.

During the preprocessing 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. Let $\lambda = 1/\epsilon \cdot \log^2 u$ and let $v_0 = \lambda^{1/8}$. 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 ball 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

[3] The original paper by van Emde Boas gives a space bound of $O(u)$, but this can be easily reduced to $O(n)$ by using randomized dynamic hashing; in this case the time bounds are in the expected sense.

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

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

Case 0. $|P| = 1$: store the point p from P.

Case 1. $|P| > 1$ and $v > v_0$:

1. split $[v]^2$ into v square blocks B_{ij}, $i, j = 0 \dots \sqrt{v} - 1$, each of size $\sqrt{v} \times \sqrt{v}$
2. recursively build data structures D_{ij} with parameters $(B_{ij}, P \cap B_{ij})$
3. if $\lambda < \sqrt{v}$ then for each B_{ij}:
   (a) split B_{ij} into λ^2 square blocks C_{kl}, $k, l = 0 \dots \lambda - 1$
   (b) compute a set $S_{ij} = \{(k, l) : C_{kl} \cap P \neq \emptyset\}$
   (c) recursively build a data structure E_{ij} with parameters $([\lambda]^2, S_{ij})$
4. compute a set $H = \{(i, j) : B_{ij} \cap P \neq \emptyset\}$
5. recursively construct a data structure F with parameters $([\sqrt{v}]^2, H)$

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

1. compute the matrix M of size $v \times v$ such that for every $i, j = 0 \dots v - 1$ we have $M_{ij} = 1$ iff $(i, j) \in P$
2. store the concatenated rows of the matrix M as one bit vector W of length v^2 (notice that $v^2 = O(\sqrt{\log u})$ and so W fits into one word)

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

SEARCH(q):

Case 0. $|P| = 1$: trivial.

Case 1. $|P| > 1$ and $v > v_0$:

1. check if $S = \emptyset$, where S is a 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 call the data structure H with p as a parameter and return the result
3. otherwise (if $S \neq \emptyset$)
   (a) let B be the block in S closest to p and let r be its distance to p
   (b) choose the set S' containing all blocks from S with distance from q in $[r, r + \sqrt{2}\sqrt{v}]$
   (c) if E_{ij}'s exist then for each block $B_{ij} \in S'$ call the data structure E_{ij} with q as a parameter and let r' be the distance to the closest *actual* point returned (say p); otherwise, assign $R = S'$ and go directly to step (e.ii.)
   (d) if $r' \geq 1/\epsilon \frac{\sqrt{v}}{\lambda}$, return p
   (e) otherwise
      i. let R denote all (at most 4) blocks B_{ij} such the distance from q to B_{ij} is at most r'
      ii. for all $B_{ij} \in R$ run the data structures D_{ij} with parameters q and output the closest point returned.

Case 2. $|P| > 1$ and $v = v_0$: call procedure *bitmap*.

Figure 2. The search procedure for approximate nearest neighbor.

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 3. 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 $[v]^2$.

This completes the description of the procedures. Now we proceed with the proofs. Due to the lack of space we present only the proof of correctness.

Correctness. The correctness of the 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 these data structures contains any other E_{ij} structure (as $v = \lambda$ and therefore $\lambda > \sqrt{v}$). Therefore, each recursive call invokes either data structures D_{ij} (step (3.e.ii) or H (step (2)). For simplicity we can assume that in step (3.e.ii) 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 length 3; each call in-

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 word (prepared during the preprocessing) which is a concatenation of $O(v_0)$ words A_r, where each A_r, for $r = 1 \ldots O(v_0)$, is a "sliced" (as in case of W) bitmap of a ball centered at q with radius r
   (c) find the first bit in A set to 1 and return the point p corresponding to that bit

Figure 3. The Bitmap procedure (case 2, Figure 2)

vokes 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 side of B_{ij} grows by a factor 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 (3d). The 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_{kl}.

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.*

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 [15] to obtain a $(1+\epsilon)$-approximation in the following way. During the preprocessing, for $r = 1, (1+\epsilon), (1+\epsilon)^2 \ldots$ (i.e. for $O((\log u)/\epsilon)$ different values of r) and for each database point p we build a data structure which for any query q checks (approximately) 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 [15] 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 [15]; 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 [15]. 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 $O(\log n)$. We defer the description to the full version of this paper.

4 Complexity of Segment Quadtree

Fat objects. A planar convex object c is *α-fat* if the ratio between the radii of the balls s^- and s^+ is at least α, where s^+ is the smallest ball containing c and s^- is a largest ball that is contained in c (for fat, non-convex objects, the definition is more involved. Here we consider only convex fat shapes). 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 bounded by a constant s_0. Let $\lambda_s(n)$ denote the maximal length of (n, s)-Davenport-Schinzel sequences (see [2]). It is known that $\lambda_s(n)$ is almost linear in n for any constant s. It is shown in [9], (see also [12]), that the number N of vertices of $\partial \cup \mathcal{C}$ is only $O(\lambda_s(n) \log^2 n \log\log n)$, for an appropriate constant s.

Theorem 4.1 *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: The idea of the proof is to charge each leaf of T to a vertex v of $\partial \cup \mathcal{C}$, in a way that v is not charged "too many" times. The details of the proof appear in the full version of the paper. □

For the rest of the discussion on quadtrees we use the following notation. Let $[u]$ denote the integer interval $\{0 \ldots u-1\}$. Let S denote a shape consisting of a union of cells of the grid $[u]^2$. Let $D(S) = S \oplus B_r$ denote the dilation of S with a ball B_r of radius $r > 0$ (note that $D(S)$ is not necessarily a shape in $[u]^2$). Let $T = T(S)$ denote the quadtree representing S.

It is known that since S consists of $\leq n$ disjoint convex objects, then $\partial D(S)$ contains $O(n)$ vertices. See [17]. 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 technique as in the proof of Theorem 4.1.

Lemma 4.2 *Let $\tilde{T}$ be a segment quadtree constructed for $\partial D(S)$. Then the number of leaves in $\tilde{T}$ is $O(n \log u)$.*

The above discussion shows that the dilated shape can be stored efficiently in a quadtree. The size of an un-compressed segment quadtree is $\log u$ times the number of its leafs, while the size of the compressed quadtree is linear with the number of leafs. In Section 6 we show how the dilated shape can be computed in a (optimal) linear time, with respect to the input region quadtree. 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).

5 A point-location data structure

Theorem 5.1 *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. In case 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 literature [25].

In case 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 $\{i \dots j\} \subset [t]$ 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 the interval $[t]$. One can observe that any interval I can be decomposed into $O(\log t)$ primitive intervals in a tree-like fashion, by finding the largest primitive interval J in I, removing it from I and recursing on $I \setminus J$. One can observe that such a decomposition is unique.

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 behaviour of binary search procedure, and thus concludes the proof of Theorem 5.1. □

Claim 5.2 *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 $[i \dots j]$.*

6 Quadtree Dilation in Linear Time

Given a shape S stored as a region quadtree, $T = T(S)$ as in Section 4, We present algorithm for computing $D(S)$ in $O(n)$ time. The algorithm consists of the 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 $D(S)$. Each of these parts consists of several steps which are briefly described bellow. The complete details can be found in the full paper [1].

We will 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, $d \in \{up, down, left, right\}$, and let $I = e_{down}$ and $x_0 \in I$ denote the lower edge of R and a point on it[4]. 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 4). 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 I. (see Figure 4). 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 term.

Lemma 6.1 *Let v and u be vertices of T, such that R_v and R_u are interior disjoint, then*

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

[4] the same apply to all other three directions

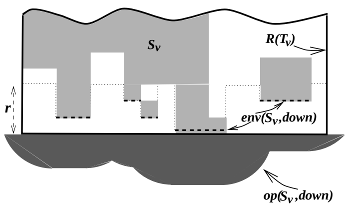

Figure 4. Left: the envelope $env(R_v, down)$ of the shape S_v is marked by the dashed line, and its dilation under the down-edge (hence its outer-path, $op(R_v, down)$) is filled in black.

where d is the direction at which R_u refers to R_v. That is, only $\mathrm{env}(R_v, d)$ 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 include R_v; one can observe that r_0 is the side of the largest tree block having this property. A *large block/leaf* is any tree block/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) leafs 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 leafs 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 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

Recall that 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, denoted as the outer path of v. The outer path at each direction $d \in \{up, down, left, right\}$ can be computed separately. By Lemma 6.1, the outer path at direction d can be computed from the envelope $env(R_v, d)$ alone. The algorithm scan the envelope and calculate its (local) expansion. The new arc/line segment is then added to the current outer path by searching for the intersection between the two, then adding the new outer segment and deleting the segment/s covered by it, if any. The detailed linear time algorithm for this step is given in the full paper.

To compute $env(R_v, down)$ we first compute a partition of $I_v = e_{down}(R_v)$ into intervals, and then we compute the y-location associated with each interval and construct the envelope as a list. To find the partition of I_v, we project T_v, the sub-quadtree rooted at v, into a binary tree, T'_v, as follows: Let the node $w' \in T'_v$ denote the projection of $w \in T_v$. 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).

The algorithm traverses T_v in a DFS order, and simultaneously constructs (and traverses) T'_v. This is called a *projection* algorithm, as all the nodes v_i having (1) the same depth, 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. The partition of I_v is provided by the intervals associated with the leafs of T'_v. During the projection process, each node $w' \in T'_v$ maintains $y_{min}(w')$ - the smallest y value (down-edge) among all black leafs projected to it (if any). The associated y location at the interval corresponds to a leaf w' is given by the smallest y_{min} value stored in the nodes along the path from the root v' down-to to the leaf w'.

The Merging (Zipping) Process

To explain the dilation merging algorithm we will need the following definition: For any large block v, let $s^{(i)}_{v,d}$ 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 5). A corridor is represented by a double-linked list of all the atom blocks it contains and the large leafs 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 takes place in large gray blocks. First, the block, v, builds its data structure using its child's data structures. Then it processes the *active zipper area* - those parts of its sons' corridors which are not included in its own corridors. These corridor parts are found near the edges shared by two sons (see Figure 5). 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).

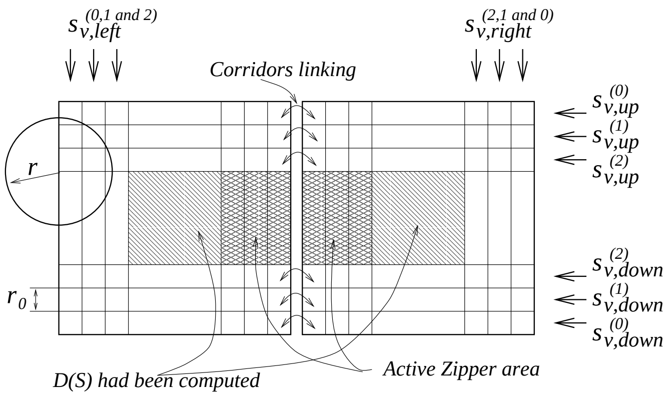

Figure 5. Two large blocks are shown, during their merging (zipping) process.

The main invariant of the algorithm is the following: when the algorithm exits a vertex v towards its parent, all vertices of $D(S_v)$ (the algorithm output) whose distance to the boundary of R_v is at least $3r_0$ have already been reported (see Figure 5). More precisely, before leaving v it computes the vertices of $D(S_v)$ which lie in those parts of the corridors of its four sons that do not intersect with its own corridors (note that v's corridors are a subset of the union of its sons' corridors). It also updates relevant data structures required for further steps of the algorithm. This process is called *merging* or *zipping* of blocks.

7 Storing the dilated shape as a quadtree

In this section we show that it is convenient to store the output of the dilation algorithm of Section 6 as a segment quadtree.

Theorem 7.1 *Let S be a planar shape given as a quadtree T consists on n nodes, all defined on the integer grid $[u]^2$, 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 ([6, 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 [22]. 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 6, whose output can be arranged (that is, directly reported) as a collection of closed curves of $\partial D(S)$ - the boundary of $D(S)$. The segment quadtree representation 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. We start to 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 $\tilde{T}$, then we split v, while updating the hash table, the net pointers, and keeping the tree balanced (which may require further splits, but, as quoted above, it sums up to a linear number of additional nodes). On the other hand, if an arc γ of C intersects with the boundary of R_v and 'leaves' this node, 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.

Second, we have to label all the empty nodes in the tree. Up to this point, any empty node, which does not contain any line segment or arc, does not know if it is inside or outside $D(S)$. This can easily be found using an algorithm similar to connected-component-labeling. It traverses the tree, while adding all the non-empty leafs to a queue. Then, it pops one leaf at a time, uses it to label its (yet unlabelled) empty neighbors, and adds only those newly-labelled 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 each boundary path of $D(S)$. Their number cannot exceed $O(n)$ (and is assuringly much smaller). Using the technique from Theorem 5.1, 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 4.2 we know that $m = O(n \log^2 u)$, we thus have proven Theorem 7.1. $\square$

The size bound 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 7.2 *Given n* disjoint *objects, a compressed quadtree representation of their union can be computed in time $O(N \log\log u)$, where N is the size of the resulted quadtree.*

For proof see [1]. By applying the technique from the above Lemma to Theorem 7.1 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.

Finally, we address the probelm of point location in dilated shape. It is shown that in order to efficiently answer such queries we do not even need to calculate the dilated shape.

Theorem 7.3 *Let S be a planar shape given as a region quadtree T consisting of n nodes defined on the integer grid $[u]^2$, and let r be a given radius. 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 $D(S)$ in expected time $O(\log\log u)$.*

Proof: Only a brief proof is give. The data structure consists of two basic parts. The goal of the first part is to find whether q lies in S itself, for which we use the structure of Theorem 5.1. If the query point is not in S, then we need to check if it falls in the dilated region. The second part encodes the outer paths from the dilation, using the corridors structure. The algorithm uses an additional hash table to allow fast access to the corresponding atom block, and a van Emde Boas tree is used to find if the point is inside the region defined by the outer path inside the atom block.

$\square$

For proof see [1].

8 Discussion

The techniques developed for the linear time dilation computation can be extended to show that segment quadtree representing the dilated shape in grid $[u]^2$ has size $O(N \log^2 u)$ and can be constructed in $O(N \log^2 u)$ time; a compressed version of this quadtree has size $O(N \log u)$ and can be constructed in $O(N \log u \log\log u)$ time. Thus given a shape represented by a quadtree and a parameter r, we can build a data structure of size $O(N \log u)$ in time $O(N \log u \log\log u)$ which in $O(\log\log u)$ time checks if a given point is within distance r from the shape.

References

[1] Amir, A., Efrat, A., Indyk, P., Samet, H. Efficient regular data structures and algorithms for location and proximity problems. *manuscript* (www.graphics.stanford.edu/~alon/regdata.html).

[2] Agarwal, P., and Sharir, M. Davenport-Schinzel sequences and their geometric applications. In *Handbook of Computational Geometry*, J.-R. Sack and J. Urrutia, Eds. Elsevier Science Publishers B.V. North-Holland, Amsterdam, 1998.

[3] Ang, C. H., Samet, H., and Shaffer, C. A. A new region expansion for quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence 12*, 7 (July 1990), 682–686. (also *Proceedings of the Third International Symposium on Spatial Data Handling*, Sydney, Australia, August 1988, 19–37).

[4] 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 Algorithms* (1994), pp. 573–582.

[5] P. Beame, F. Fich, "Optimal Bounds for the Predecessor Problem", STOC'99, to appear.

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

[7] 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 (April 1992), 253–280.

[8] Edelsbrunner, H., Guibas, L. J., and Sharir, M. The complexity and construction of many faces in arrangements of lines and of segments. *Discrete Computational Geometry 5* (1990), 161–196.

[9] Efrat, A. The complexity of the union of (α, β)-covered objects. 1998. *Proceedings 15 Annual Symposium on Computational Geometry*, 1999, to appear.

[10] Efrat, A., and Itai, A. Improvements on bottleneck matching and related problems using geometry. In *Proceedings of the Twelfth Annual ACM Symposium on Computational Geometry* (Philadelphia, May 1996), pp. 301–310.

[11] 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* (1997), pp. 297–396.

[12] 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), pp. 104–112.

[13] 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* (Baltimore, Maryland, 14–16 May 1990), pp. 1–7.

[14] Har-Peled, S., Chan, T. M., Aronov, B., Halperin, D., and Snoeyink, J. The complexity of a single face of a Minkowski sum. In *Proceedings of the Seventh Canadian Conference on Computational Geometry* (Quebec City, August 1995), pp. 91–96.

[15] P. Indyk, R. Motwani, "Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality", *STOC'98*, pp. 604-613.

[16] 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.

[17] 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.

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

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

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

[21] Ramkumar, G. D. An algorithm to compute the Minkowski sum outer-face of two simple polygons. In *Proceedings of the Twelfth Annual ACM Symposium on Computational Geometry* (Philadelphia, May 1996), pp. 234–241.

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

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

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

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

[26] Yang, H.-T., and Lee, S.-J. Optimal decomposition of morphological structuring elements. In *Proceedings of the International Conference on Image Processing* (1996), pp. 1–4.