Technion - Israel Institute of Technology
Computer Science Department

Dilation and Quadtrees - Theoretical and Practical Results

by

Arnon Amir, Alon Efrat and Hanan Samet

CIS Report #9626 December 1996

Dilation and Quadtrees — Theoretical and Practical Results

Arnon Amir* Alon Efrat† Hanan Samet‡

Abstract

Let S be a given shape in the plane, and let B be a ball of a fixed radius. The dilation of S, denoted by $D(S)$, is the shape resulting from taking the Minkowsky sum of S and B. $D(S)$ is the region consisting of all points in the plane whose distance from some point of S is at most r. Computing the dilation of a shape is a common and important problem in the fields of robotics, image processing, computer graphics and in spatial databases where it corresponds to the spatial analog of a range query.

Letting $T(S)$ be the quadtree representation of S, an algorithm (termed *Algorithm A*) is presented for calculating $D(T(S))$ in an optimal $O(n)$ time where n is the number of blocks in $T(S)$). Algorithm A provides an analytic description of $D(T(S))$. However, in some applications, a result in the form of a quadtree is preferred. A second algorithm (termed *Algorithm C*) is presented that takes advantage of this output format requirement. It simplifies and applies some additional heuristics to Algorithm A resulting in a more practical algorithm, although one that is not asymptotically optimal which is the case for Algorithm A. This algorithm is currently implemented, and empirical results will be presented in the full paper.

1 Introduction

Given shapes S and Q, their Minkowsky sum, also known as their *dilation*, denoted by $\oplus$, is:

$$S \oplus Q = \{s + q \mid s \in S, q \in Q\}.$$

Computing the dilation of a shape is a common and important problem in fields such as robotics, assembly, geographic information systems (GIS), computer vision, computer graphics, as well as in spatial databases where it corresponds to the spatial analog of a range query. For example, in robotics, dilation is used in motion planning where the Minkowsky sum can be interpreted as a parametric description of the free space in which a robot can lie without intersecting any of the obstacles in its vicinity. See for example [3] for a (partial) list of applications. It is also used as a basic morphological operation in image processing and computer vision (e.g., document understanding). Among the algorithms for performing the dilation operation, we mention [5, 8] as well as others.

In this paper we address perhaps the simplest (but by no means the least important) variant of the dilation problem where both S and Q are planar shapes, S is given in the form of a region quadtree $T(S)$, and Q is a ball B_r of a fixed radius r. This problem also arises in many other fields. For example, suppose that we have a map and we want to find all the areas that are within 5 miles of a lake. In [1] a similar problem is addressed with the difference that that the dilation is done with a square rather than a circle. Note that dilation with a square is the spatial analog to a range query using the Manhattan metric instead

*Computer Science Department, Technion Haifa 32000, Israel *arnon@cs.technion.ac.il*

†School of Mathematical Sciences, Tel Aviv University, Tel-Aviv 69982, Israel. *alone@cs.tau.ac.il*

‡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.

of the Euclidean metric. However, dilating a quadtree with a circle is less natural, and cannot be achieved as a direct extension of this result.

Let us define $D(S) = S \oplus B_r$. It is easy to show that the complexity[1] of $D(S)$, denoted $comp(D(S))$, is linear in $comp(S)$, which is always small or equal to n, the number of blocks in $T(S)$. To see the first assertion, note that S can be partitioned into at most $comp(S)$ convex regions (e.g., using triangulation) and $D(S)$ is the union of the dilation of each of these regions. It was shown in [6] that the union of such regions has linear complexity. However, we are not aware of any linear time algorithm for this problem.

The region quadtree (e.g., [9, 10]) is a hierarchical representation of region data that is based on the recursive decomposition of a bounded image array into four equal-sized quadrants. In order to define a region quadtree, without loss of generality, assume a binary image that contains just one region where the pixels in the region are black (i.e., 1) and the pixels in the background are white (i.e., 0). If the array does not consist entirely of 1s or entirely of 0s (i.e., the region does not cover the entire array), then it is subdivided into quadrants, sub-quadrants, etc. until blocks are obtained that consist entirely of 1s or entirely of 0s (i.e., each block is entirely contained in the region or entirely disjoint from it). If the image exhibits homogeneity (i.e., it is not a checkerboard), then the quadtree can result in storage savings. Regardless of how much storage is saved, region quadtrees are also attractive because many of the algorithms for operations on images represented by region quadtrees execute in time proportional to the number of blocks in the quadtree rather than the number of pixels in the image array. This leads to considerable improvements in the execution time of algorithms such as connected component labeling (e.g., [2]).

In this paper we attempt to take advantage of the fact that the input shape is given as a region quadtree. We are interested in finding an algorithm that computes the dilation of any shape represented by a region quadtree in time proportional to the number of blocks in the quadtree. Moreover, we wish to know what kind of information, in addition to the description of shape S, is needed to obtain the dilation of S (denoted by $D(S)$) in a linear time.

There are two reasons to address this question. Quadtrees are often used for representing shapes in existing data-bases. Therefore, taking advantage of this information seems economical. The other reason is purely theoretical: we want to answer the question of *what kind of information is needed, except the description of S itself, in order to obtain a linear time algorithm for computing $D(S)$?* For instance, it could be shown that if the Voronoi diagram of S is given, then a linear time algorithm is also easily obtained[2]. Hence the question "what kind of additional information do we need, in order to compute the dilation in a linear time" appears to be interesting. In this paper we show that the order posed by the quadtree is sufficient.

In this paper we present two variants of the dilation algorithm. Let S be a shape in the Euclidean plane $\mathbb{R}^2$, whose quadtree representation is denoted as $T(S)$. In $T(S)$ each leaf is either white or black, where S is the union of the black leafs of $T(S)$. Section 2 contains some definitions. Section 3 describes an optimal algorithm, named *Algorithm A*, whose asymptotic running time is optimal in the size of the input. Algorithm A uses a general quadtree projection technique, described in Section 4. It produces an analytic description of $D(T(S))$. Section 5 presents a second version of the algorithm that produces the output in a quadtree form. It uses the same quadtree projection technique together with a simplified merging stage, that is based on some heuristics, to save time in the digitization and the quadtree building processes. Algorithm C is currently implemented, and the results of our experience will be given in the full paper. An up-to-date version of this paper can be found at *http://www.cs.technion.ac.il/~arnon/papers/qt_dilation.ps.gz*.

2 Preliminaries

Let R denote a rectangle[3], let $T = T(S)$ be a quadtree of the shape S, and let $R(T)$ denote the bounding square of the region occupied by T. For a rectangle R we refer to its four edges as the up-edge, down-edge,

[1] In this setting, the complexity of a shape S means the number of vertices and edges of S.

[2] We thank Joseph S. B. Mitchell for this remark.

[3] All rectangles and squares in this paper are axis-parallel, and are lying on the discrete grid

left-edge and right-edge. Let $d \in \{up, down, left, right\}$, say $d = down$, and let e be the edge of R which faces the d direction. Let q be a point on e, and let $N(q)$ be the closest among all points of S which are vertically above q, at a distance smaller than r from q (recall that r is the radius of dilation). We define $N(q) = r$ (alternatively ∞) if no such point exists. When considering $N(q)$ as a function of q, its graph is piecewise constant. For a shape S enclosed in a rectangle R, this graph is termed the *d-relevance* of S in R, and is denoted by $rel(S, R, d)$ (see Fig. 1). We also define $rel(R, d) \equiv rel(S_{in} \cap R, R, d)$, where S_{in} is the original input shape, given by the quadtree T_{in}. By abusing notation, we sometimes refer to $rel(R, d)$ as a polygonal path description of the piecewise-constant function $N(q)$ stored as a doubly-connected list of its vertices. Naturally, $rel(R, d)$ is defined analogously for $d =$ up, left, right.

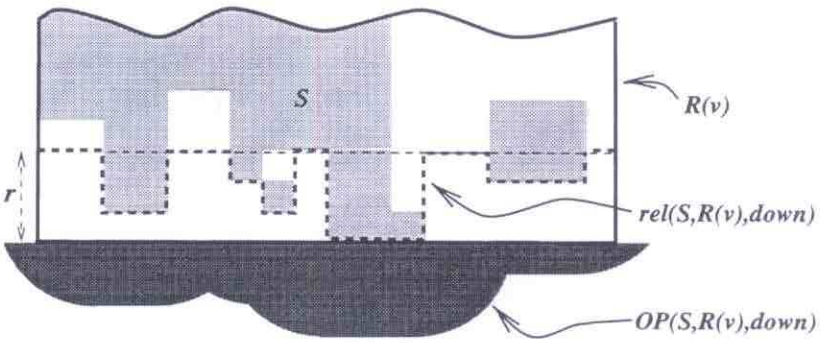

Figure 1: The relevance $rel(S, R, down)$ of the shape S is marked by the dashed line, and its dilation under this border (hence its outer-path, $OP(S, R, down)$) is filled in black.

The next lemma shows the importance of the relevance of a shape term.

Lemma 2.1 *Let S be a shape, enclosed in a rectangle R, and let R' be another rectangle, disjoint to R, and lying completely in the d-direction of R, for $d \in \{up, down, left, right\}$, then*

$$D(S) \cap R' = D(rel(S, R, d)) \cap R'$$

That is, only $rel(S, R, d)$ counts in terms of influencing R' by the dilation of S.

Let r_0 be chosen so that

$$\sqrt{2} \cdot r_0 = \sqrt{2} \cdot 2^d \leq r < \sqrt{2} \cdot 2^{d+1} = 2\sqrt{2} \cdot r_0$$

for an integer d (see Fig. 2). We say that a block $v \in T$ is an *atom block* if the edge-length of $R(v)$ is exactly r_0. By this choice of r_0 it is guaranteed that an atom block v that contains a black block will become completely black after the dilation with B_r. Therefore, the result of $D(v)$ is a simple shape, that has no holes (note that an atom block may be black, white or gray). A *large block* is a block larger than an atom block.

We use the term *effect* of S in a region R to mean to $D(S) \cap R$ — the part of the dilation of S found in R. Let the *effective neighborhood* of a block $v \in T$ be the union of all the atom blocks that can effect any point in v. A crucial observation is that if q is some point in the plane, then the dilation of only a constant number of atom blocks surrounding q might effect q (no more than three atom blocks to each side). This observation is the basis of the efficiency of our algorithm.

For a set S, enclosed in a rectangle R, let the *outer-path* of R in the d-direction, denoted $\mathrm{OP}(S, R, d)$, describe the boundary of $D(S \cap R)$ found in the half-plane at the d side of R (see Fig. 1). The outer-path

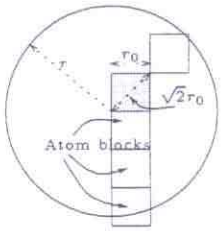

Figure 2: The selection of r_0, given the dilation radius r.

is a *polyarc* that consists of a list of straight segments and arcs of radius r. By Lemma 2.1, the outer-path depends only on the relevance, and can be computed by $\mathrm{OP}(S, R, d) = \partial(rel(S, R, d) \oplus B_r)$. As before, we sometimes use shorthand notation and define $\mathrm{OP}(R, d) \equiv \mathrm{OP}(S_{in}, R, d)$.

For a node v of $T(S)$, let $T(v)$ denote the subtree rooted at v. We say that v is a *grey block* if it contains both white and black regions.

3 Algorithm A: Linear Time Dilation.

In this section, we describe an algorithm whose input is a shape S_{in}, represented as a quadtree T_{in}, and computes $D(S) = S \oplus B_r$ in time $O(n)$, where n is the number of blocks in T_{in}. The output of this algorithm is an analytic description of $D(S)$, that is, each black shape is described by a list of arcs and line segments along its boundary.

3.1 Basic Steps of the Algorithm

We first partition the region $R(root(T))$ into vertical slabs, denoted s_i, $i = 1, 2, 3, \ldots$, each of width r_0. The slabs are ordered from left to right, such that each atom block of T fits into exactly one slab. An *interesting slab* is a slab which **might** contain a vertex of $D(S)$ in its vicinity (in a distance less than $3r_0$). We avoid processing the non interesting slabs. Let $\mathcal{L}[i]$, $i = 1, 2, \ldots, k\}$ denote the list of all k interesting slabs, and let γ_i be the vertical right border of slab s_i.

The algorithm consist of the following steps

1. **Projecting:** Project T into a binary tree T'. Each slab is stored in a node of T' as a linked list of its atom blocks, $b_{i,j}$, ordered from bottom to top. In addition, each atom block $b_{i,j}$ stores two pointers to the closest large blocks which are above and below $b_{i,j}$ (if any).

2. **Computing $\mathcal{L}$:** Compute the list of interesting slabs, $\mathcal{L}$ (Fig. 3(left)). For each interesting slab, compute the intersection region, $s_i \cap R$, of any large black block R that intersects with s_i and has a corner in s_i, or has an atom block $b_{i,j}$ (in s_i) within a distance less than $6r_0$ above or below R.

3. **Computing the dilation in s_i** By the effective neighborhood observation, the dilation result inside slab s_i only depends on the content of s_i and (at most) three slabs to its left and three slabs to its right (Fig. 3(right)). The effects of the left side and of the right side on s_i are separately evaluated using their relevances.

4. **Slab sweeping:** The effects of atom blocks $b_{i,j}$ and of large black blocks inside s_i are evaluated while merging their results with the effects of its left and right sides. After this stage, each slab s_i contains all vertices of $\partial D(S) \cap s_i$.

5. **Slabs tailoring:** Every slab s_i is tailored to s_{i+1} by connecting adjacent black (white) regions along the vertical border γ_i. The resulting tailored boundaries of $D(s)$ are reported to the output. Small white regions of $\mathbb{R}^2 \setminus D(S)$ (white holes), which are fully contained in (the interior of) s_i, are also reported. The tailoring process also takes care of these parts of large black blocks that have not been processed in the above stages (to be explained later).

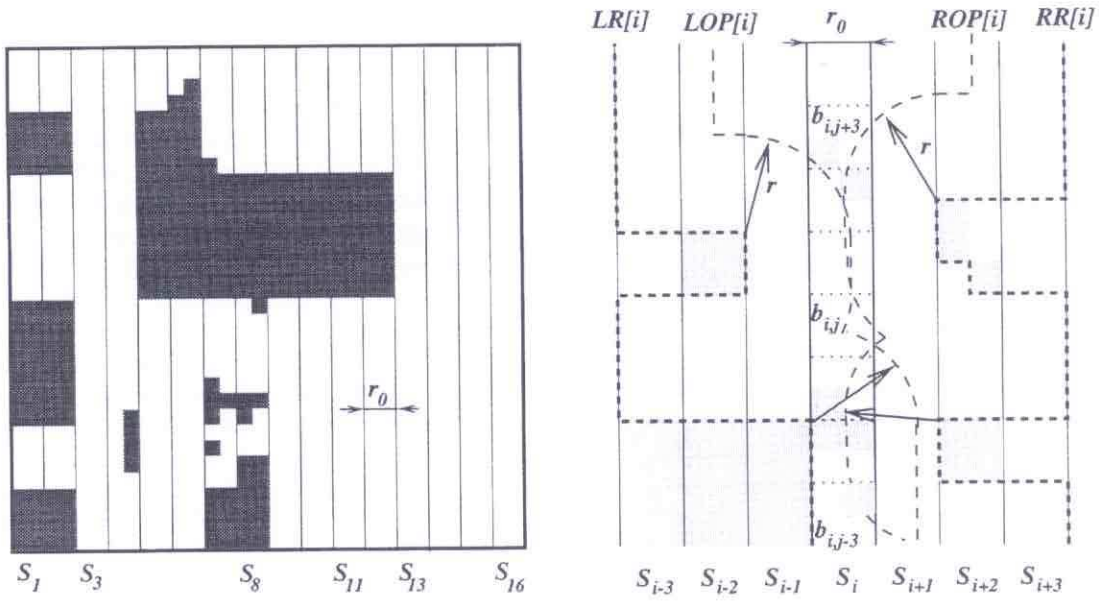

Figure 3: Algorithm A: The image region is divided into slabs of width r_0. Only the *interesting* slabs (here all 16 slabs, except s_{16}) are processed (left). The dilation is first computed inside every interesting slab, s_i, from bottom to top, in atom blocks $b_{i,j}$ (right). The slab s_i merges the effect of the three slabs on its left and on its right. The relevance on each side, $LR[i], RR[i]$ and the outer-paths, $LOP[i], ROP[i]$ (the relevance dilated with B_r) are shown in thick and in thin dashed lines, respectively.

Special care should be taken at every stage of the algorithm in order to accomplish the task in a linear time. A new data structure, denoted *a projected quadtree*, is therefore introduced. Two instances of the projection procedure are used in stages 1 and 3.

The reader should be aware of the complexity of processing the large black leafs. Although dilating them is as easy as dilating smaller blocks, the boundary of their dilated area might intersect with a large number of smaller regions, and might have a relatively high complexity, while their input complexity is always 1. Therefore, special care is taken in the description of how the algorithm processes large black blocks.

We now give more details about the algorithm. Some of the finer details of a few of the procedures are not given here. Instead, they are given in Section 4.

3.2 Projecting T and Computing $\mathcal{L}[i]$

We split the tree T into atom blocks, which are white, black or gray nodes of size r_0 in T, and into *large blocks*, namely those leafs whose width is larger than r_0. We use the projection tree $T' = proj(T)$ which is described in Section 4.1, to scan T. Each atom block that we encounter is projected onto the x-axis. As a result, we receive for each interesting slab s_i (containing any black block), a corresponding linked list $\mathcal{L}[i]$ of all the atom blocks (and some parts of large black blocks, as defined above) that it contains. The elements

in every linked list $\mathcal{L}[i]$ point to the gray atom blocks found in s_i. The atom blocks are denoted $b_{i,j}$ and are ordered from bottom to top. To avoid confusion, observe that not all the slabs are interesting, and that $\mathcal{L}[i]$ includes only the interesting ones.

After finishing the projection stage, we scan $\mathcal{L}$, and insert few more interesting (empty) lists, so that at the end of this process, the list $\mathcal{L}$ includes for every non-empty slab s_i all of its six neighboring slabs, $s_{i-3}, s_{i-2}, s_{i-1}, s_{i+1}, s_{i+2}$ and s_{i+3}. It is easy to observe that the number of slabs inserted is at most n.

3.3 Computing the Outer-Paths in a Slab s_i

This procedure is applied separately to each one of the slabs, s_i, $i = 1, 2, \ldots, k$. First, we compute the *left relevance*, LR[i], defined as $rel(\mathcal{L}[i-3] \cup \mathcal{L}[i-2] \cup \mathcal{L}[i-1], d)$. Similarly, we compute the *right relevance* RR[i], defined as $rel(\mathcal{L}[i+1] \cup \mathcal{L}[i+2] \cup \mathcal{L}[i+3], d)$. Both are shown in Fig. 3(right). The detailed description of this procedure, which uses the quadtree projection technique, is given in Section 4.3.

Next we compute the *left-outer-path* LOP[i] of LR[i] (which is $s_i \cap (\mathrm{LR}[i] \oplus B_r)$), consisting of arcs and straight segments (Fig. 3(right)). Similarly, we define (and compute) ROP[i] from RR[i]. They are stored as a y-monotone polyarc which is a pseudo-polygonal path. The exact procedure by which we evaluate LOP[i] from LR[i] is described in Section 4.4.

Next, we compute for each atom grey block $b_{i,j}$, its relevances in the downward and upward directions and for each relevance we compute the corresponding outer path (using the same procedures). In addition, we also compute for each of these outer-paths its horizontal decomposition. Assume C is a (downward or upward) outer path, The *horizontal decomposition* of C is defined as follows (see Figure 4). From each vertex or an extreme point of C, we construct two horizontal rays, one to the left and one to the right, until each ray hits an edge of C, or one of two auxiliary vertical lines that we add at $x = -\infty$ and $x = \infty$. This yields a trapezoidation Z of C.

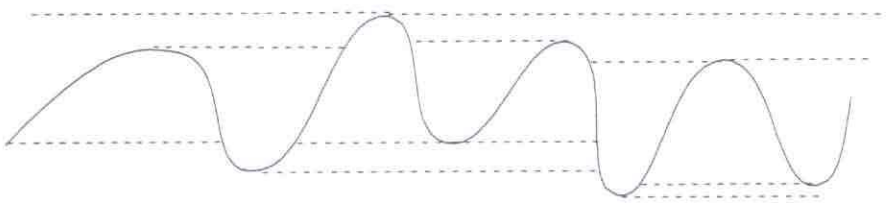

Figure 4: The *horizontal decomposition* of a curve is a trapezoidation of the plane.

Let DHD[ij] denote the horizontal decomposition of the outer-path in the downward direction of $b_{i,j}$. Similarly, we define UHD[ij]. Before we proceed further, let us describe how we compute in linear time the horizontal decomposition of C. We first find for each vertex of C the edge (or vertex) horizontally to its left. We use a stack S which is initially empty. We scan C from its leftmost point to the right. When the scan meets a vertex of C, we push it into S. When we reach an edge (an arc of a circle) f containing a point q whose y coordinate value is equal to that of the point t on top of S, we add the segment $\bar{qt}$ as a part of the trapezoidation, and pop t from S. We repeat this operation in the left direction. All trapezoids are stored in a DCEL data structure (see [7] for a description of this data structure). It is easy to see that the total time needed for this process is $O(n')$, where n' is the complexity of C.

3.4 Slab Sweeping

Given the outer-paths found in s_i, we can compute $D(S) \cap s_i$. This is a merging-like process that should evaluate the boundary of the union of all black regions inside s_i. For each slab s_i, we perform a variant of an

upward 'segment-sweep' inside s_i, where e is the horizontal sweep-segment. During this sweep, we record the location at which e intersects LOP$[i]$ and ROP$[i]$. We also know at each instance of time the corresponding place in the list $\mathcal{L}[i]$ — that is, in which block $b_{i,j}$ the segment e is found. Hence we know which three blocks are immediately below and which three blocks are immediately above the block $b_{i,j}$ (Fig. 3(right)). It is easy to verify that no other block in s_i is found in the effective neighborhood of $b_{i,j}$. However, we do want to know at each instance of time which edges of $rel(b_{i,j+1}, down)$ and $rel(b_{i,j+1}, up)$ are intersected by the segment e. Not having this information would require sorting them in order to find the next intersection point, which we cannot afford. However, using the their horizontal decompositions, UHD$[i, j-1]$ and DHD$[i, j+1]$, sorting is not needed.

Typically, we need to find the vertices of the boundary of the union of the regions which is bounded by at most four outer-paths. Assume e is currently in $b_{i,j}$. Surely, if $b_{i,j}$ is a grey or a black (atom) block of T, then no boundary point of $D(S)$ appears in this block. Hence assume that $b_{i,j}$ is white, and is intersected by (some of) the paths C_l, C_r, C_u, C_d, where $C_l = \text{LOP}[i], C_r = \text{ROP}[i]$, C_u is the outer-path of the relevance of the (at most) three blocks currently above $b_{i,j}$, and C_d is the outer-path of the relevance of the (at most) three blocks currently below $b_{i,j}$.

Actually, it is enough to find all intersection points between each pair C_i, C_j for $i, j \in \{l, r, u, d\}$, $i \not= j$. Observe that C_i is given in a sorted order of its vertices. Therefore, the intersection points of C_l with C_r and of C_u with C_d are obtained by a simple merging procedure. However, calculating the intersections of C_l (say) with C_d or with C_d is a bit tricky (Figure 5). This is where we use the horizontal decompositions of C_d (C_u). We obtain these points by traversing C_l (C_r) while maintaining at each instance of time the identity of the trapezoid in which we are at each step of passing through the horizontal decomposition.

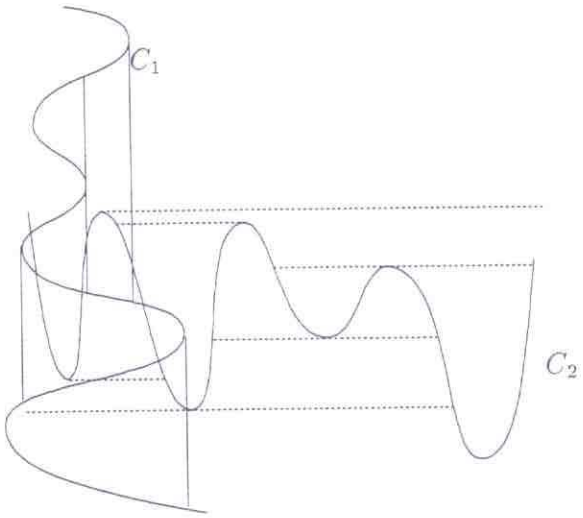

Figure 5: The merging of two orthogonal outer-paths is done using the horizontal decomposition of C_2 (see text).

First we locate the trapezoid containing the leftmost point of C_l, and than walk along C_l while locating at every instance the trapezoid in which we are in. This enables us to find all intersection points of C_l and C_d.

Observe that while rolling a point along C_l, we might enter the same trapezoid of C_d an arbitrary number of times. However, since C_r, is y-monotone, all but (perhaps) one of these entrances must be made through the part of the boundary which is a part of OP$[i, j]$ — that is, not through one of the edges that we added

in the horizontal decomposition process. Each such entrance must therefore be a vertex of the region $D(P)$, where

$$P \equiv S_{in} \bigcap (s_{i-3} \cup s_{i-2} \cup s_{i-1}) \cup (b_{i,j-3} \cup b_{i,j-2} \cup b_{i,j-1}).$$

Appealing again to the same theorem of [6] used above, the complexity of $D(P)$ is proportional to the complexity of P, which is $O(n)$ when summed over the course of the algorithm.

3.5 Tailoring the Slabs Results

The last stage of the algorithm combines all the parts of the results found in the slabs. The output is a list of closed paths around black and white regions, each given as list of vertices, arcs, and line segments. Each slab s_i computed in the previous stage a list of all the vertices in it, $\partial D(S) \cap s_i$, sorted from bottom to top. However, we must provide for three different types of possible vertices.

1. Vertices of a small white region which is entirely included inside the slab s_i (obviously there are no such small black regions after the dilation).
2. Vertices which lie on a short boundary segment so that its next vertex is in the immediately left (right) neighboring slabs, s_{i-1} and s_{i+1}.
3. Vertices which lie on a long boundary segment so that its next vertex is in a far slab. This can only occur as a result of dilating a large black block.

We tailor two slabs at a time, along the vertical line γ_i which separates them, starting from γ_0. While visiting s_i, the vertices of type 1 are immediately reported as close path(s) to the output. The vertices of type 2 are "merged" with those of s_{i-1}, along γ_i, in upward order. However, vertices of type 3 have no corresponding vertex in s_{i-1}; instead, it is in an older slab, say s_j, where $j < i$. These two vertices, one from s_j and one from s_i, use the node $v \in T$ — the large black leaf that corresponds with this long border segment to communicate. While visiting s_j, the vertex stores its identity in v, and while visiting s_i the second vertex uses it to report the long line segment.

The total number of vertices in all slabs is linear in n. By performing this merging step we visit each vertex only twice, and therefore the time needed for tailoring is linear with n.

4 Quadtree Projection and its Applications

4.1 Projecting a Quadtree into a Binary Tree

First, we describe the procedure for the case where $R = R(v_r)$ is a square region occupied by a sub-quadtree $T = T(v_r)$ under node v_r (the root of T). Next, we add a few words on the more general case, where R may be any given rectangle. Recall that in a quadtree T, each node v is associated with the square block $R(v) = I_v \times J_v$ that is occupied by it, where I_v and J_v are intervals in $\mathbb{R}$, on the x and y axes, respectively. For $d = down$ (or $d = up$) the binary tree represents the projection of $T(v_r)$ on the x axis. Let T' denote a binary tree. Each node in T' is associated with an interval $I_{v'}$ in $\mathbb{R}$, on the x axis. Any quadtree node $v \in T$ is projected onto one node v' in the binary tree T', the one which is associated with the same interval $I_{v'} = I_v$.

For a node $v \in T$, let $ul(v), ur(v), ll(v), lr(v)$ denote the upper-left, upper-right, lower-left and lower-right, children of v, respectively. The *projection path*, $ppath(v)$, from v_r to v in T is defined as a series of steps t_i made along the path, where $t_i \in \{ul, ur, ll, lr\}$. Let v'_r denote the root of T', which is the projection of v_r. The path from v'_r to v' in T' is a series of steps $t'_i \in \{l, r\}$, and is defined by translating the projection path, step by step. Every ul or ll step is translated to an l step, and every ur or lr step is translated to an r step. We call T' the *projected tree* of T in a region R to direction d, and denote it by $T' = proj(T, R, d)$.

Since the translation of the projection path is careless about the up and down parts of the steps, all the nodes in T which are associated with the same interval I_v over the x axis are projected to the same node $v' \in T'$. Let $l_{v'}$ denote the list of all black leafs in T that are projected to node $v' \in T'$. For reasons to be explained later, we associate with every black leaf $v \in T$ a value $y_{min}(v)$, defined as the endpoint of J_v — that is, the y coordinate value of the lowest point in $R(v)$. For each $v' \in T'$ we associate the value $y_{min}(v') = min\{y_{min}(u) \ : color(u) = black, \ u \in l_{v'}\}$, which is the minimal $y_{min}(u)$ value among all black leafs $u \in T$ which are projected to v'.

Now, let us explain how we construct T'. We visit T in the order (for each $v \in T$) $ul(v), ll(v), ur(v), lr(v)$. We also maintain a pointer v' inside T'. Initially, both v points to $root(T)$ and v' points to $root(T')$. Assume recursively that we are at a node $v \in T$, and at its projected node $v' \in T'$. While moving from $v \in T$ to one of its children $u \in T$, we do the following: If u is a left (right) child of v, we move from v' to u' which is the left (right) child of v'. If u' is null, then we create this node, and set its two children and $l_{u'}$ as nulls. We insert u as the last element of the list $l_{u'}$. Once our traversal returns from u to its parent v, we also move from u' to v'. Clearly, T' is the projected tree of T, and its construction is performed in time $O(n)$, where n is the number of nodes in T.

We use T' for two purposes: to compute $\mathcal{L}$ (given in Section 4.2 and used in Algorithm A) and to compute the relevances (given in Section 4.3 and used in Algorithms A and C). The projection algorithm is summarized in Fig. 6, including the additional computation of $y_{min}(v')$. Its running time is linear in the size of the input quadtree. In the more general case, that is used only in Algorithm C, the region R may be any rectangle. In such a case, the procedure starts at a node v such that $R \subseteq R(v)$, and recursively visits only the sons which are associated with a region that intersects with R. In this case, the running time is linear with the number of visited nodes.

```
1. Initialize an empty binary tree T' = ∅.

2. Traverse T, starting from the root v_r, and keep
   following the projected path in the binary tree.
   For each node v ∈ T {
      if (v' ∉ T') {
         insert(v', T')
         y_min(v') = ∞
         l_v' = ∅
      }
      if (color(v) = BLACK) {
         l_v' = {l_v', v}
         y_min(v') = min{y_min(v), y_min(v')}
      }
   }
```

Figure 6: Projecting a quadtree T to a binary tree T'.

4.2 Computing the Slabs List $\mathcal{L}$

. In Section 3.1, we needed a procedure to generate $\mathcal{L}$, the list of projected atom blocks of T_{in}. For this, we generate the projected tree T', and then use the lists $l_{v'}$ of the atom nodes (that is $|I_{v'} = |I_v| = r_0$). We use the projection algorithm described above, with the two following modifications:

- For each atom node $v \in T$, listed in $l_{v'}$ in the projected tree T', we also store a pointer from $l_{v'}$ back to v in T.
- For each atom node in $l_{v'}$ we store a pointer to the lowest large black leaf above it.

The projection tree has the property that for each $v' \in T'$, the y-intervals J_v of the blocks that are listed in $l_{v'}$ appear in an increasing y order. By traversing T' in in-order and scanning only those members of lists $l_{v'}$ that contain atom nodes, we have that each $l_{v'}$ is exactly one list $\mathcal{L}[i]$ corresponding to a slab s_i. Notice that the in-order nature of our projecting traversal ensures that if v_1 is visited before v_2 where both v_1 and v_2 are nodes of the same level of T', then I_{v_1} precedes I_{v_2} along the x axis.

The first modification is easy to handle, while inserting v to the list $l_{v'}$. The second modification is maintained in a similar way to the calculation of $y_{min}(path(v'))$ in the projection process.

4.3 Computing the Relevance $rel(R(T), T, d)$

The binary tree T' implies a recursive binary partition of the x axis into intervals $I_{v'}$. Here we are interested in the partition defined by the leafs of the tree. The depth of a leaf determines the length of its interval, and the path from $root(T')$ defines its location (analogous to a one-dimensional "quadtree"). Let $path(T', v') = \{v_1 = root(T'), v_2, \ldots, v_k = v'\}$ denote the path in T' from the root to a node $v' \in T'$. Let $y_{min}(path(v')) = min\{y_{min}(v_i) \ : \ i = 1, \ldots, k\}$ denote the minimal value of $y_{min}(v_i)$ that is encountered along $path(v)$. For a point $x_0 \in \mathbb{R}$, let $rel(R, down)(x_0)$ denote the y-value at the intersection of $rel(R, down)$ with the vertical line $x = x_0$. The next lemma is easily established.

Lemma 4.1 *Let $x \in \mathbb{R}$, and assume $x \in I_{v_x}$, where v_x is a leaf of T'. Then*

$$rel(r, down)(x) = y_{min}(path(v_x))$$

Proof: Let $v \in T$ be the lowest black leaf (if any) in T which intersects the vertical line at x. Let $y = y_{min}(v)$ be the lowest y coordinate value of J_v. Its projected node $v' \in T'$ is found on the path $path(v')$. Furthermore, $y_{min}(v') = y$, and it is easy to show that $y_{min}(path((v')) = y_{min}(v')$. If there is no such a black leaf $v \in T$, then $y_{min}(v'_i) = \infty \ \ \forall \, v'_i \in path(v'_r, v')$ □

This lemma suggests the following simple procedure for calculating $rel(R, down)$, which uses a depth-first traversal (DFS) on T'. Maintain the connected list $rel(R, down)$ which is initially empty. Traverse the projected binary tree T'. At each node v' calculate $y = y_{min}(path(v')) = min\{y_{path}, y_{min}(v')\}$, where y_{path} is received from v''s father. If v' is a leaf, then connect the interval $I_{v'}$ to the list $rel(T, R, down)$, and assign to it the y-value. Otherwise, transmit y as the new y_{path} value to its two sons. The pseudo-code for this procedure is given in Fig. 7.

Observe that in the application denoted above, we need to compute the relevance for several neighboring regions (sub-quadtrees), instead of a single one, whose blocks lie one above the other. However, for $d = left$ or $d = right$ this only calls for the concatenation of the relevances of each of the distinct blocks, and for $d = up$ or $d = down$ all blocks should use the same projected tree.

It should be clear that the execution time of the algorithm is linear in the size of T'. For computing the slabs list, $\mathcal{L}$, this procedure is applied once over T_{in}. For computing $rel(s_i, d)$, the sub-quadtrees under each atom block in each slab are being used, 7 times each (each slab pays once as any one of $s_{i-3}, \ldots, s_{i+3}$). Therefore, the execution time of the procedure is linear.

4.4 Computing the Outer-Path $\mathbf{OP}(R, d)$

Let R be a rectangle, containing a shape S, and let d be a direction. Let ℓ be the horizontal line containing the lower edge of R. We need to compute the *outer-path* $\mathrm{OP}(R, d)$, which is the part of $\partial D(S \cap R)$ above ℓ (see Fig. 1). For simplicity, we assume that $d = down$. By lemma Lemma 2.1, this outer-path can be evaluated from $rel(S, R, d)$ — the relevance of S in region R to direction d, found by the algorithm from

```
find_rel(v', y_path) {
    y = min{y_path, y_min(v')}
    if v' is not a leaf {
        find_rel(v'.l_son, y)
        find_rel(v'.r_son, y)
    }
    else /* v' is a leaf */
        output(I_v', y)
}
```

Figure 7: Evaluating $rel(T, R, d)$, the relevance of quadtree T, in a region R at direction $d = down$.

Section 4.3. The complexity of the relevance is linear in the complexity of S in R. To perform this task in time proportional to the complexity of $rel(R, d)$, we need the following lemma, taken from [4];

Lemma 4.2 *Assume the rectangle R is the disjoint union of R_l and R_r, where R_l, R_r are rectangles, and R_l lies to the left of R_r. Then $OP(R_l, down)$ and $OP(R_r, down)$ intersect at most once.*

Recall that $rel(R, down)$ is a piecewise constant function of x. We scan $rel(R, down)$ from left to right, and process one segment (constant-y piece) at a time. Let α, α' denote the left and right points of the current segment, e, respectively. Let R_α denote the part of R which is to the left of a vertical line passing through α. Let $UE(\alpha)$ denote $\mathrm{OP}(rel(R_\alpha, down))$. Assume that we have already computed $UE(\alpha)$, and let β be the rightmost point of $UE(\alpha)$. We seek $UE(\alpha')$. For this, we only need to find the intersection point q (if it exists) of $UE(\alpha)$ and $OP(e, R, down)$. Next, we remove the part of $UE(\alpha)$ lying between q and β, and concatenate the "new" region of $UE(\alpha')$ which lies between q and the rightmost point of $\mathrm{OP}(e, R, down)$.

Finding and deleting the part $q\beta$ from $UE(\alpha)$ is achieved as follows: Let $q' = \beta$. We roll the point q' on $UE(\alpha)$ in the left direction as long as q' is in $OP(e, R, up)$. q is the point at which q' leaves $\mathrm{OP}(e, R, up)$. Lemma 4.2 ensures that no other intersection point exists. The time needed for computing $UE(\alpha')$ (after $UE(\alpha)$ has been determined) is proportional to the complexity of $q\beta$. Since each part in the region was created only once, and was removed only once, and since the number of elements in $UE(\alpha)$ is proportional to the complexity of $rel(R_\alpha, d)$, the execution time over the course of the procedure is proportional to the complexity of $rel(S, R, down)$.

5 Algorithm C: Applying Practical Heuristics.

Like Algorithm A, Algorithm C also takes as its input a shape S_{in} as a quadtree T_{in}, and evaluates its dilation with a circle of radius r. However, while Algorithm A produces an analytic description of $D(S)$, Algorithm C produces a quadtree T_{out} of $D(S)$. The conversion from an analytic description into a quadtree requires a digitization and quadtree building process. It can be done using the chain-code to quadtree algorithm, found in [9, 11], in linear time with respect to the total perimeter of all shapes in $D(S)$. However, in practice, there are other considerations besides asymptotic complexity such as time constants and code simplicity, as well as some heuristics that may also reduce the overall computational time. In particular, we would like to avoid the two non-simple (yet linear time) merging processes of polyarcs described in Section 3: the segment sweeping inside every slab and the slabs tailoring. Algorithm C addresses these points on the basis of the following observations:

(a) Every atom block becomes black in the output tree.

(b) There could be a large overlap between expanded neighboring regions.

(c) All the effective dilation events occur along the borders between gray atom blocks (or large black blocks) and white blocks (see Fig. 8).

(d) There is no need to digitize the parts of the analytic description which are covered by an existing or a new black block.

An extreme example of the effect of (a) is an input image which contains a large number of isolated black pixels. Most of the image space will become black by applying this criterion. From (b) we conclude that in order to save on merging operations, it is better to expand few large areas than the many atom blocks that were defined earlier. The problem with large areas is how to deal with non simple shapes, that contain holes. As a result, Algorithm C does not divide the image into disjoint processing regions (like the slabs in Algorithm A). Instead, it works along *maximal boundary segments*, which are segments of maximal length, that have white leafs along one side (the *white side*) and black (or gray atom blocks) on the other side (the *black side*) (see Fig. 8). The maximal boundary segments can be found in a linear time using the algorithm in Appendix A (or the quadtree to chain-code algorithm found in [9]). The main difference from algorithm A is the replacement of the two merging processes, namely the segment sweeping and the slabs tailoring, with one simpler merging process (thereby not requiring the use of slabs at all). For this we introduce the concept of a *mixed quadtree*, where every leaf can be black, white, or a collection of analytic curves (more details follow). Hence the new merging process is done between mixed quadtree nodes. As a result, there is no need to explicitly compute the analytic description in any stage of the process. Moreover, no digitization needs to be applied on an explicit form of the dilated shape (as there is no such form). We could suggest to dilate, digitize, and convert every atom block to a subquadtree, and then to merge the many resulting subquadtrees using a simple quadtree merging procedure. However, this would have a heavy price in terms of requiring many extra digitizations as observed in (d). Therefore, we delay the digitization of analytic curves into black and white leafs as long as possible. In fact, in many cases this digitization is eliminated by performing it simultanuesly with the merging process as explained in Appendix B. Algorithm C has four main stages:

1. Copy T_{in} to T_{mixed} while replacing all subtrees of atom gray blocks with new atom black leafs (Fig. 8(b)).

2. Find all maximal black/white boundary segments in T_{mixed} (Fig. 8(c)).

3. For each maximal boundary segment, s_i:
   3.1 Find its $rel(T_{in}, s_i, d)$, where d is the black side of s_i (Fig. 1)
   3.2 Evaluate its outer path at the white side of s_i (Fig. 1)
   3.3 Merge the result into T_{mixed}

4. Convert T_{mixed} into a black/white quadtree, T_{out}.

Note that stages 3.1 and 3.2 use the same procedures as Algorithm A: the quadtree projection algorithm given in Section 4.1, the relevance algorithm given in Section 4.3, and the outer-path algorithm given in Section 4.4.

6 Discussion

It is already known that the complexity of the union of dilated convex shapes is linear in the number of shapes [6]. However, no algorithm that we know about could find that result in a linear time. In this paper a linear time dilation algorithm is described, for the special case where a shape given as the union of the black leafs of a quadtree is dilated with a circle.

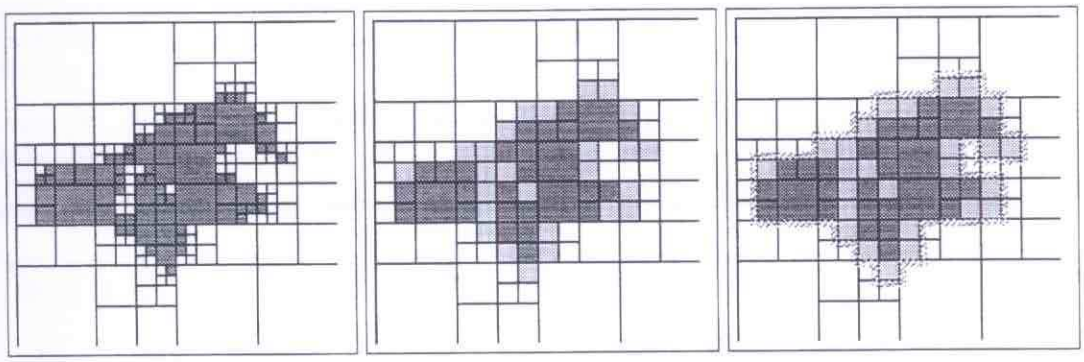

Figure 8: Algorithm C expands only the parts necessary for expansion (here $r = 6$). An example input quadtree, shown in (a), is divided into atom gray blocks and large black/white blocks in (b). In Algorithm C, only the marked borders are considered (C), their relevances and the outer-paths are being computed (the rest are merged with black boxes).

Algorithm C is currently implemented, and the results of our experience will be given in the full paper. An up-to-date version of this paper can be found at *http://www.cs.technion.ac.il/~arnon/papers/qt_dilation.ps.gz*.

There are a number of directions for future research. The main goal is to find a more general dilation algorithm for dilating with structuring elements other than a ball, and for shapes given using other spatial data structures. This should enable us to gain a better insight on the question of "what kind of additional information do we need, in order to compute the dilation in a linear time". As shown here, the order posed by the quadtree is sufficient.

References

[1] 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).

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] PREPARATA, F. P., AND SHAMOS, M. I. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[8] 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.

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

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

[11] WEBBER, R. E., AND SAMET, H. Linear-time border-tracing algorithms for quadtrees. *Algorithmica 8* (1992), 39–54. (also University of Maryland Computer Science TR 2309).

Appendix A: Finding the Maximal Boundary Segments

The algorithm takes as its input a black/white quadtree T, and evaluates a list of maximal boundary segments, as defined earlier in Section 5. The time of the algorithm is linear in the number of nodes in T.

Let $J(v) = \{j_1, j_2, ...\}$ denote a list of the black/white color change points along the boundary of the square region $R(v)$ occupied by $T(v)$. The size of $J(v)$ is linear with the size of the tree, $O(|T(v)|)$. (see Fig. 9(b)). Let $HS(j_i)$ denote the half segment of the internal black/white border that is orthogonal to the boundary of $B(v)$ and that terminates at the junction j_i. Let $FS(v)$ denote the list of full segments found inside $B(v)$. These are all the black/white borders which have both ends inside $B(v)$.

The algorithm traverses the nodes of T in a DFS order. If the visited node v is a leaf, then $J(T(v))$ is a list of its four borders, $HS()$ does not exist, and $FS(v) = \emptyset$. For any non-leaf node v, the algorithm has already computed $J(), HS()$ and $FS()$ for all four sons. The union of all four son's $FS()$ lists forms the initial $FS(v)$ list. $J(v)$ and $HS(v)$ are just the concatenation of the non-adjacent parts of v's sons $J()$ and $HS()$ lists, respectively. In addition, four potential new half-segments are located at the junctions of each two adjacent sons (see Fig. 9(c)). The algorithm continues by merging $J()$ of each two brothers along their adjacent border. While doing that, each pair of colinear half segments that have the same color order is joined into one segment. If a segment has no endpoints in $J(v)$, then it is moved to $FS(v)$.

At the root node v_r of the input quadtree, all the remaining half segments are moved from $HS()$ to $FS(v_r)$. The output of the algorithm is $FS(v_r)$. In order to get a complete chain code out of this algorithm, one should keep for each segment in $FS(v)$ two links to its neighboring segments.

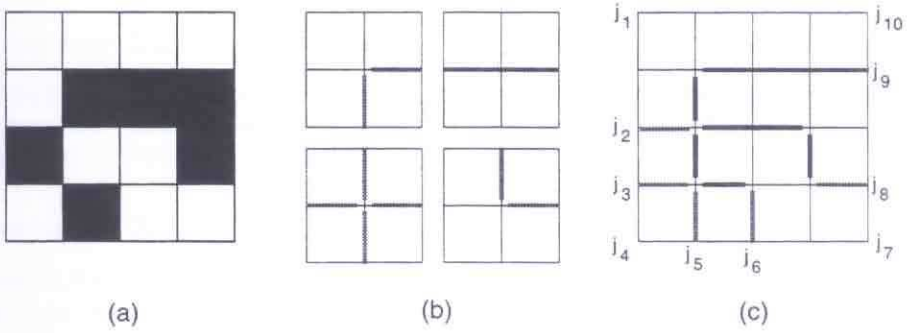

Figure 9: Evaluating the black/white border: The quadtree under node v is shown in (a). The recursive process collects all sons results (b) and evaluates $FS(v)$ (black segments) and $HS(v)$ (gray segments), as shown in (c).

Appendix B: Mixed Quadtrees (MQT)

Let a *mixed quadtree (MQT)* denote a quadtree that have three types of leafs: black (opaque) leafs, white (transparent) leafs, and sketch leafs. A *sketch* leaf contains a union of black regions over a white background. Each black region is a simple shape, bounded by a polyarc. In the scope of this work, the MQT supports the four following operations:

- Convert a quadtree into a MQT (trivial).
- Split a sketch leaf into four sons.
- Merge two MQTs.
- Convert a MQT into a regular quadtree.

The MQT is used as an intermediate data structure during the merge process. Its main advantage is that merging is postponed until all the relevant information is collected. A white leaf might receive many parts of outer paths from its neighborhood. However, it is enough to find one such part that covers the entire leaf to avoid the merging of all the others. Therefore, merging is postponed until all parts have been collected.

The space complexity of a MQT is the number of its black and white leafs plus the total complexity of its sketch leafs. The complexity of a sketch leaf is the number of line segments and arcs in it and is not bounded (which is also the case for the PM and PMR quadtrees [9]).

B.1 Convert a Quadtree into a MQT

A quadtree is a special case of MQT. Therefore, no action should be taken for this operation.

B.2 Splitting a Sketch Leaf

The *split* procedure splits a sketch leaf v into four leafs (sons). This is done by splitting each polyarc in v into four sons. The time is linear with respect to the complexity of the sketch. Note that each son may become a black leaf if it is completely covered by one of the sketches black regions, or become a white leaf if it does not intersect any of the sketch's black regions. Otherwise, it becomes a sketch leaf. In some cases, all four sons may become black, and the sketch vanishes. We do not test whether or not the union of all parts of a sketch cover the leaf area.

B.3 Merging two Mixed Quadtrees

The merge of MQTs is a MQT of the union of all their black regions. The merging algorithm is based on the traditional recursive quadtree merging algorithm (found in [9]), except for the case of merging a sketch leaf with a gray node, in which the sketch node should be split.

B.4 Converting a MQT into a Quadtree

In order to convert a MQT back into a quadtree, each of the sketch leafs must be converted into a subquadtree. This is done by applying the split() algorithm recursively, until the sketch vanishes or its size becomes one pixel. In the last case, a black/white decision is taken upon digitization (e.g., point sampling).