An Optimal Algorithm for One-separation of a Set of Isothetic Polygons, With Extensions to Higher Dimensions

Amitava Datta* Kamala Krithivasan† Thomas Ottmann*

Abstract

We consider the problem of separating a collection of isothetic polygons in the plane by translating one polygon at a time to infinity. The directions of translation are the four isothetic (parallel to the axes) directions, but a particular polygon can be translated only in one of these four directions. Our algorithm detects whether a scene is separable in this sense and computes a translational ordering of the polygons. The time and space complexities of our algorithm is $O(n \log n)$ and $O(n)$ respectively, where n is the total number of vertices of the polygons in the scene. Our algorithm is optimal with respect to both time and space. The best previous algorithm in the plane for this problem had complexities of $O(n \log^2 n)$ time and $O(n \log n)$ space. We also present extensions of our algorithm to higher dimensions. The time and space complexities of our algorithm in three dimensions are $O(n\sqrt{n} \log n)$ and $O(n\sqrt{n} \log n)$ respectively. In a dimension $d \geq 4$, the time and space complexities are $O(dn^{d/2} \log n)$ and $O(dn^{d/2} \log n)$ respectively. The time complexities of our algorithm in a dimension $d > 2$ improve the previous best complexities for this problem by a factor of $O(\log^2 n)$.

*Institut für Informatik , Universität Freiburg, Rhein Straße 10-12, D-79104 Freiburg i. Br., Germany. *e-mail:* {datta,ottmann}@informatik.uni-freiburg.de

†Department of Computer Science and Engineering, Indian Institute of Technology, Madras 600 036, India. *e-mail:* kamala@iitm.ernet.in

1 Introduction

The problem of collision-free motion of objects in the presence of obstacles has many applications in *computer graphics*, *CAD-CAM systems* and *robotics*. The objects to be moved varies from one application to another. There has been considerable work on moving line segments, polygons and polyhedra in two and three dimensional spaces in the presence of obstacles. These problems are collectively called *motion planning* problems. In this paper, we are interested in a different kind of motion planning problem called *separability* problem. This class of problems is interesting from the point of view of two potential application areas. In the first case, we have a collection of objects and we are interested in finding out whether the objects can be separated one by one from the collection by collision free motion. In the second case, the problem arises in machine assembly. A robot has to assemble or disassemble a composite machine part by moving elementary machine parts through collision free motion [5]. Reif [11] has proved that this problem is PSPACE-hard in its most general form. However, when only translational motion is allowed, this problem is tractable and has been studied extensively in recent years. The problem can be stated in the following form. Suppose, CP is a composite part having M simple components $P_1, P_2, \ldots, P_M$ and each of the simple components is a polygon with n vertices. Is it possible to disassemble CP by translating one simple part at a time? Usually, two related problems are considered in this context. The first is the *detection* problem, where it is to be decided whether a composite part can be disassembled or not. In the *determination* problem, a disassembly sequence is computed such that the composite part can be disassembled following this sequence of translations.

We first discuss the existing results when only a single direction of translation is allowed and each simple part should be translated in this common direction. Guibas and Yao [4] have shown that a disassembly sequence for a set of M rectangles (resp. convex n-gons) can be computed in $O(M \log M)$ (resp. $O(Mn + M \log M)$) time. Ottmann and Widmayer [8] gave an algorithm for translating a set of line segments. They also gave an alternate and simple solution for the problem studied in [4]. It is clear that for convex composite parts, a disassembly sequence exists for any fixed direction. However, in case of non-convex parts, there may not exist any direction of disassembly. This happens if a subset of the parts are interlocked. Nurmi [6] has shown that both the detection and determination problems for a composite part with non-convex components can be solved in $O(Mn\log(Mn))$ time. Toussaint [12] also gave alternate solutions for the disassembly problem. Recently, Nussbaum and Sack [7] have presented matching lower and upper bounds for detecting and determining a disassembly sequence in a single direction. Their algorithm runs in $\Theta(Mn + M\log M)$ time.

In the isothetic domain,i.e., when the polygons have sides parallel to the axes, the first work on separation is due to Guibas and Yao [4]. Chazelle *et al* [1] first considered the problem of disassembling a set of isothetic polygons. They considered separating the polygons in directions parallel to the axes (i.e., isothetic directions). They considered two problems related to this. The first problem is called the *iso-separability* problem. In this case, the direction of translation is unique. Chazelle *et al* [1] presented an optimal $O(n\log n)$ time and $O(n)$ space algorithm for the iso-separability problem. Here, n is the total number of vertices of the polygons. In the *one-separability* problem, all the four isothetic directions of translation are allowed. However, a particular polygon can be translated only once in one of these four directions. Notice that, this problem is more general in the sense that more than one direction of translation is allowed. Chazelle *et al* presented an $O(n\log^2 n)$ time and $O(n\log n)$ space algorithm for deciding whether a collection of isothetic polygons is one-separable. They could also compute a separating sequence within the same complexity. Recently, Devine and Wood [3] have investigated this one-separability problem in 3 and higher dimensions. In three dimensions, the algorithm of Devine and Wood [3] works in $O(n\sqrt{n}\log^3 n)$ time and $O(n\sqrt{n}\log n)$ space. In dimensions $d \geq 4$, their algorithm runs in $O(dn^{d/2}\log^3 n)$ time and $O(dn^{d/2}\log n)$ space.

In this paper, we present the first optimal algorithm for deciding one-separability and computing a separation order of the polygons in two dimensions. Our algorithm runs in $\Theta(n\log n)$ time and $\Theta(n)$ space. Though the basic strategy of our algorithm is similar to that in Chazelle *et al* [1], our data structure is completely different and we expect that this data structure will be useful in other applications. If the input scene is not one-separable, our algorithm finds a maximum subset of the polygons which is one-separable. As a result of our two dimensional algorithm, we also improve all the time bounds established by Devine and Wood [3] for one separation in 3 and higher dimensions. In 3 dimensions, our algorithm requires $O(n\sqrt{n}\log n)$ time and $O(n\sqrt{n}\log n)$ space. In dimensions $d \geq 4$, the complexity of our algorithm is $O(dn^{d/2}\log n)$ time and $O(dn^{d/2}\log n)$ space. Hence, our improvement in running time is a factor of $O(\log^2 n)$ compared to the algorithms in [3].

The rest of this paper is organized as follows. In Section 2, we give an informal description of our algorithm. We describe our data structure in details in Section 3. The algorithm for one-separability in two dimensions is presented in Section 4. In Section 5, we present the extension of our algorithm to higher dimensions. Finally, we conclude in section 6 with some comments and open problems.

2 An Informal Description

The basic strategy of our algorithm is similar to that in Chazelle *et al* [1]. We first construct four balanced binary trees in the four isothetic directions and then try to peel these trees, i.e., try to free the polygons by deleting free edges from these four trees. A high level description of the algorithm follows.

Step 1. We sort the vertical (resp. horizontal) edges of all the polygons in ascending and descending order according to x (resp. y) coordinates.

Step 2. We construct four skeletal balanced binary trees as mentioned before.

Step 3. We insert the vertical edges into the eastern (resp. western) tree in decreasing (resp. increasing) x order, by sweeping a line from maximum (resp. minimum) x to minimum (resp. maximum) x coordinates. Similarly, we insert the horizontal edges in the northern (resp. southern) tree in decreasing (resp. increasing) y order.

Step 4. We attempt to peel the trees by starting with any one of the four trees and deleting the free edges in that tree. After deletion of a free edge, we check if any polygon is freed. If such a polygon P_i exists, we delete the edges of P_i from the other three trees and in the process find other polygons which are freed due to the freedom of P_i. If no polygon is free in a particular tree, we go to the other trees one by one and repeat this process. This process is continued until one of the following cases happen. In the first case, all the polygons get freedom and the scene is one-separable. In the second case, at some stage of the algorithm neither an edge nor a polygon can be freed from any of the four trees. In this case, the scene is not one-separable.

The main difference between our approach and that in Chazelle *et al* [1] is the data structure we use. In [1], the data structures for the four trees are segment trees [10]. As a result, there are certain difficulties in maintaining freedom information of the edges. The main problem is to check whether an edge is free in a particular isothetic direction. An edge may be blocked by a large edge which encloses it, or by a small edge which is enclosed by it. Though it is relatively easy to check the first case using segment trees, the second case is more difficult. Chazelle *et al* kept a secondary structure at each internal node of the segment tree to check this case. This secondary structure is a priority queue. As a result, the search becomes a two level search, first in the nodes of the segment tree and then in the priority queues associated with the nodes of the segment tree. This leads to a complexity of $O(n \log^2 n)$. Another problem with this data structure is that segment tree requires $O(n \log n)$ space since each edge may be present in $O(\log n)$ nodes.

Our data structure is a simple balanced binary tree. However, we maintain additional fields in the internal nodes to facilitate our search. The number of such additional fields is constant at every internal node and we do not need any two level structure. As a result, we are able to achieve the optimal complexities of $O(n \log n)$ time and $O(n)$ space for the one-separability problem.

The details of the data structure and the algorithm are given in the next two sections.

3 The Data Structure

As mentioned earlier, our algorithm is based on manipulations of the four trees in the four directions. The data structure for each direction is a balanced binary tree augmented with additional fields in their internal nodes. These four trees are called the *eastern*, *western*, *northern* and *southern* trees. These four trees are used to detect freedom in the $+x$, $-x$, $+y$ and $-y$ directions. We use the abbreviations E_tree, W_tree, N_tree and S_tree for these four trees. In our discussion, we assume that the polygons are in general position i.e., no two vertical (resp. horizontal) sides have the same x (resp. y) coordinate. Our algorithm can be modified easily in the degenerate case. In the following, we discuss the fields in the eastern tree and its construction. All our discussion hold for the other three trees in a similar way.

In the E_tree, the y coordinates of the end points of the vertical intervals are stored in the leaves in sorted order from left to right. The left-most (resp. right-most) leaf stores the minimum (resp. maximum) y coordinate. For each vertical edge e_i, we denote the leaves storing its two end points by $l(e_i)$ and $r(e_i)$, where $l(e_i)$ (resp. $r(e_i)$) denotes the end point with lower (resp. higher) y coordinate. Under the general position assumption, each leaf stores only two end points.

We keep some additional fields at each node of E_tree. In the rest of this section, we will refer to the E_tree as T to simplify the notation.

3.1 Fields in the internal nodes

Let n_i be an internal node of T and $T(n_i)$ be the subtree of T rooted at n_i. The two children of n_i are denoted as $left(n_i)$ and $right(n_i)$. We denote the parent of n_i by $parent(n_i)$. We say that an edge e_k is *blocked* in the eastern direction in the subtree rooted at n_i, if there is an edge e_j such that e_j has at least one end point in $T(n_i)$, e_j and e_k overlap and $x(e_j) > x(e_k)$. If there is no such edge e_j, we say that e_k is *unblocked* in $T(n_i)$. The additional fields at n_i are the following.

$min(n_i)$: A field which stores the coordinate of the left-most (the least y coordinate) leaf in $T(n_i)$.

$max(n_i)$: A field which stores the coordinate of the right-most (the highest y coordinate) leaf of $T(n_i)$.

$L\_high(n_i)$: A field which stores an edge which is completely contained and unblocked in the left subtree of n_i. If there is no such edge e_i, then $L\_high(n_i) = null$. Otherwise, let e_k be the edge with the highest x coordinate among all edges satisfying the two conditions above. Then, $L\_high(n_i) = e_k$.

$R\_high(n_i)$: A field which stores an edge which is completely contained and unblocked in the right subtree of n_i. If there is no such edge e_i, then $R\_high(n_i) = null$. Otherwise, let e_k be the edge with the highest x coordinate among all the edges satisfying the two conditions above. Then, $R\_high(n_i) = e_k$.

$LL\_cross(n_i)$: A field which stores an edge e_i whose left end point $l(e_i)$ is in the left subtree of n_i, but right end point is not in the left subtree of n_i, i.e., $min(left(n_i)) \leq l(e_i) \leq max(left(n_i)) < r(l_i)$. Also, the part of e_i in the left subtree of n_i is unblocked. If there is no such edge e_i, then $LL\_cross(n_i) = null$. Note that only one edge can satisfy the above conditions.

$LR\_cross(n_i)$: A field which stores an edge e_i whose right end point is in the left subtree of n_i, but whose left end point is not in that subtree, i.e., $l(e_i) < min(left(n_i))$ and $min(left(n_i)) \leq r(e_i) \leq max(left(n_i))$. Also, the part of e_i in the left subtree of n_i is unblocked. If there is no such edge e_i, then $LR\_cross(n_i) = null$. Again, only one edge can satisfy the above conditions.

$RL\_cross(n_i)$: A field which stores an edge e_i whose left end point is in the right subtree of n_i, but whose right end point is not in that subtree, i.e., $min(right(n_i)) \leq l(e_i) \leq max(right(n_i))$ and $r(e_i) > max(right(n_i))$. Also, the part of e_i in the right subtree of n_i is unblocked. If there is no such edge e_i, $RL\_cross(n_i) = null$. Only one edge can satisfy the above conditions.

$RR\_cross(n_i)$: A field which stores an edge e_i whose right end point is in the right subtree of n_i, but whose left end point is not in that subtree, i.e., $l(e_i) < min(right(n_i))$ and $min(right(n_i)) \leq r(e_i) \leq max(right(n_i))$. Also, the part of e_i which is in the right subtree of n_i is unblocked. If there is no such edge e_i, then $RR\_cross(n_i) = null$. Only one edge can satisfy the above conditions.

The definitions of the fields in an internal node are illustrated in Figure 1.

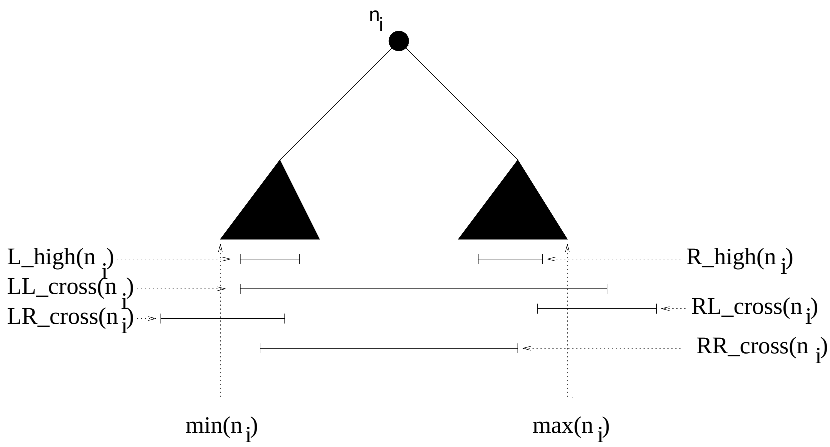

Figure 1: Illustration of the fields at an internal node n_i. If an end point of an edge crosses the base of a triangle denoting a subtree, it is assumed that the end point is outside the subtree. Also, it is assumed that at a time only one of the edges is present and it is unblocked.

3.2 Fields in the leaf nodes

In our data structure, each leaf contains two end points from two edges e_j and e_k. This is because, the vertical and horizontal edges appear alternately along the polygon boundary. Note that, due to the general position assumption, no leaf can contain more than two end points. We denote the x coordinate of a vertical edge e_i by $x(e_i)$. Also note that the fields L_high and R_high are undefined in the leaf nodes since no leaf node can contain a complete edge. For determining the fields in the leaf nodes, we have the following cases.

$Case(1)$**:** The leaf node n_i has the left end points of two vertical edges e_j and e_k. Let these two end points be $l(e_j)$ and $l(e_k)$.

If $x(e_j) > x(e_k)$ then
$LL\_cross(n_i) := RL\_cross(n_i) := e_j$
else
$LL\_cross(n_i) := RL\_cross(n_i) := e_k$;
$LR\_cross(n_i) := RR\_cross(n_i) := null$;

$Case(2)$**:** The leaf node n_i has two right end points, i.e., $r(e_j)$ and $r(e_k)$.

If $x(e_j) > x(e_k)$ then
$LR\_cross(n_i) := RR\_cross(n_i) := e_j$
else
$LR\_cross(n_i) := RR\_cross(n_i) := e_k$;
$LL\_cross(n_i) := RL\_cross(n_i) := null$;

$Case(3)$**:** The leaf node n_i has one left end point $l(e_j)$ and one right end point $r(e_k)$.
$LL\_cross(n_i) := RL\_cross(n_i) := e_j$;
$LR\_cross(n_i) := RR\_cross(n_i) := e_k$;

3.3 Construction of the trees

In this section, we discuss the construction of the four trees. We discuss only the construction of the E_tree in details and indicate briefly how the other trees can be constructed in a similar way. For simplicity of notations, we again refer to the E_tree as T.

We sort the vertical edges of the polygons in order of decreasing x coordinates and sweep the set of polygons from right to left (i.e., from higher x coordinate to lower x coordinate). The y coordinates of the end points of the edges are inserted in T. In a first sweep, we construct a skeletal structure for the tree T. In other words, we construct the binary tree with its internal nodes and leaves, but without any of the vertical edges inserted in it. However, we set up the fields $max(n_i)$ and $min(n_i)$ at every internal node n_i. This can be easily done by knowing the coordinates of the two extreme leaves of the subtree rooted at such a node n_i.

In a second sweep, we insert the vertical edges in a right to left sorted order. In this sweep, we also establish the fields in all the nodes. This can be easily done from the information of $max(n_i)$ and $min(n_i)$ and the max and min fields of the two children of n_i. We only discuss how the field $LL\_cross(n_i)$ can be set up at an internal node n_i. The computation of the other fields is also similar. As mentioned earlier, $LL\_cross(n_i)$ is an edge whose left end point is contained in the subtree rooted at $left(n_i)$, but the right end point is not in the subtree rooted at $left(n_i)$. Also, such an edge is unblocked in the subtree rooted at $left(n_i)$ in the eastern $(+x)$ direction. While doing the second sweep, if an edge has its left end point in the subtree rooted at n_i, but its right end point is not in this subtree, we update the field $LL\_cross(n_i)$ only if this field is currently empty. Since we are sweeping from higher x to lower x direction, once we find such an edge, we need not update this field again. Since the end points of the edges are stored in the leaves, this updation can be done for all the nodes along the insertion path in $O(\log n)$ time. So, at the end of the second sweep, we have inserted all the vertical edges of the polygons in T and we have established all the fields in the internal nodes as well as in the leaves. While inserting an edge in T, in $O(\log n)$ time all the fields along the insertion path can be updated. Hence, the complete construction of T takes $O(n \log n)$ time. A simple scene of isothetic polygons and the initial N_tree is shown in Figure 2.

3.4 Deletion of an edge and updation of the fields

In our algorithm for deciding one-separability, we only delete edges from the four trees during the peeling of the trees. We need not insert any more edges. In this section, we discuss the deletion of an edge and describe how to update the fields in the internal nodes and leaves of the tree during this deletion.

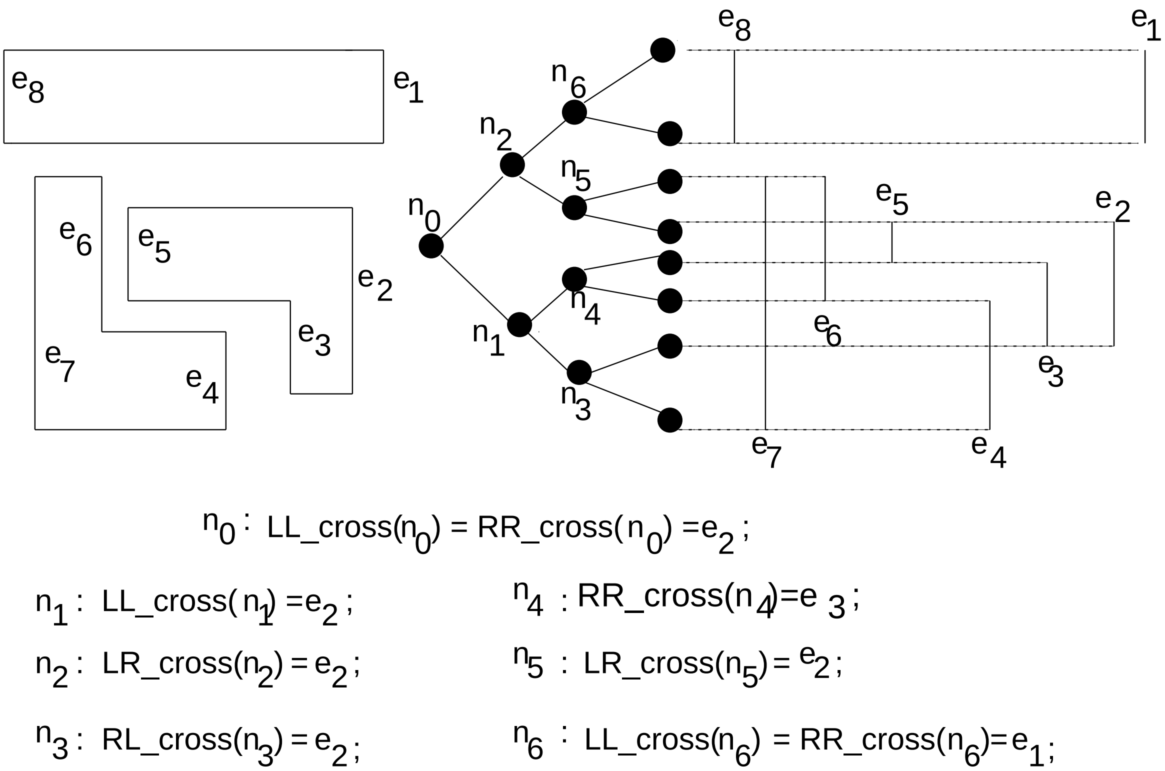

Figure 2: A simple scene of isothetic polygons and the initial N_tree. The non-empty fields at the internal nodes are shown. max and min fields are not shown.

3.4.1 Deletion of an edge

As before, the two end points of an edge e_i are denoted by $l(e_i)$ and $r(e_i)$. To delete an edge e_i, we delete the two leaves containing $l(e_i)$ and $r(e_i)$. If these two leaves are occupied by other end points from some other edges, we keep the two leaves but delete the information regarding e_i. It is ensured during the construction of the four trees that the maximum path length is $O(\log n)$. Hence, during deletion also, no path-length can exceed this bound. We assume without loss of generality that for e_i, we first delete $l(e_i)$ and then $r(e_i)$. While deleting $l(e_i)$ (resp. $r(e_i)$), we update the internal fields along the path from the leaf containing $l(e_i)$ (resp. $r(e_i)$) to the root.

3.4.2 Updation of the fields at the internal nodes

We discuss the updation of the fields in an internal node in a generic way. Suppose, we are deleting $l(e_j)$ for an edge e_j and n_i is a node along this deletion path from bottom to top. We assume that the fields upto the node n_i have already been updated. We discuss the updation of the fields in $parent(n_i)$. Let $High(e_m, \ldots, e_n)$ be a function which returns the edge with highest x coordinate among $e_m, \ldots, e_n$. There are several cases in the updation of the fields in $parent(n_i)$ when $l(e_j)$ is deleted. When $r(e_j)$ is deleted, the updations are similar and the details are omitted.

Case(1) *n_i is the left child of $parent(n_i)$*

(a) $L\_high(parent(n_i))$

If $(LL\_cross(n_i) = RR\_cross(n_i))$ then
$L\_high(parent(n_i)) :=$
$High(L\_high(n_i), LL\_cross(n_i), R\_high(n_i))$
else $L\_high(parent(n_i) :=$
$High(L\_high(n_i), R\_high(n_i))$;

(b) $R\_high(parent(n_i))$: Remains unchanged.

(c) $LL\_cross(parent(n_i))$

If $R(LL\_cross(n_i)) > max(n_i)$ then
$LL\_cross(parent(n_i)) :=$
$High(LL\_cross(n_i), RL\_cross(n_i))$
else $LL\_cross(parent(n_i)) :=$
$RL\_cross(n_i)$

(d) $LR\_cross(parent(n_i))$

If $L(RR\_cross(n_i)) < min(n_i)$ then
$LR\_cross(parent(n_i)) :=$
$High(LR\_cross(n_i), RR\_cross(n_i))$
else $LR\_cross(parent(n_i)) :=$
$LR\_cross(n_i)$

(e) $RL\_cross(parent(n_i))$: Remains unchanged.

(f) $RR\_cross(parent(n_i))$: Remains unchanged.

(g) $max(parent(n_i))$: Remains unchanged.

(h) $min(parent(n_i)) := min(n_i)$.

Case(2) *n_i is the right child of $parent(n_i)$*

(a) $L\_high(parent(n_i))$: Remains unchanged.

(b) $R\_high(parent(n_i))$:

If $LL\_cross(n_i) = RR\_cross(n_i)$ then
$R\_high(parent(n_i)) :=$
$High(L\_high(n_i), LL\_cross(n_i), R\_high(n_i))$
else $R\_high(parent(n_i)) :=$
$High(L\_high(n_i), R\_high(n_i))$

(c) $LL\_cross(parent(n_i))$: Remains unchanged.

(d) $LR\_cross(parent(n_i))$: Remains unchanged.

(e) $RL\_cross(parent(n_i))$:

If $R(LL\_cross(n_i)) > max(n_i)$ then
$RL\_cross(parent(n_i))$
$:= High(LL\_cross(n_i), RL\_cross(n_i))$
else$RL_cross(parent(n_i)) := RL_cross(n_i)$

(f) $RR\_cross(parent(n_i))$

If $L(RR\_cross(n_i)) < min(n_i)$ then
$RR\_cross(parent(n_i)) :=$
$High(RR\_cross(n_i), LR\_cross(n_i))$
else$RR_cross(parent(n_i)) := LR_cross(n_i)$

(g) $max(parent(n_i)) := max(n_i)$

(h) $min(parent(n_i))$: Remains unchanged.

3.4.3 Updation at the leaf nodes

We now consider the updation of the fields at a leaf node n_k. Under the general position assumption, n_k contains at most two end points of two edges e_i and e_j. There are several cases depending on the combination of the end points present at n_k.

Case(1) *n_k has two left end points,i.e., $l(e_i)$ and $l(e_j)$; and $l(e_i)$ is deleted*

$LL\_cross(n_k) := RL\_cross(n_k) := e_j$;
$LR\_cross(n_i)$ and $RR\_cross(n_i)$ remain unchanged, i.e., $null$.

Case(2) *n_k has two right end points, i.e., $r(e_i)$ and $r(e_j)$; and $r(e_i)$ is deleted*

$LR\_cross(n_i) := RR\_cross(n_i) := e_j$;
$LL\_cross(n_i)$ and $RL\_cross(n_i)$ remain unchanged, i.e., $null$.

Case(3) *n_k has one left and one right end point*

(a) *n_k has $l(e_i)$ and $r(e_j)$; and $l(e_i)$ is deleted*

$LL\_cross(n_i) := RL\_cross(n_i) := null$;
$LR\_cross(n_i)$ and $RR\_cross(n_i)$ remain unchanged.

(b) *n_k has $l(e_j)$ and $r(e_i)$; and $r(e_i)$ is deleted*

$LR\_cross(n_i) := RR\_cross(n_i) := null$;
$LL\_cross(n_i)$ and $RL\_cross(n_i)$ remain unchanged.

3.4.4 Correctness of the updations

Lemma 3.1 The updations discussed in Section 3.4.2 and Section 3.4.3 correctly maintains the fields in T after deletion of an edge.

Proof: The correctness of the updation at the leaf nodes is straightforward and we omit the details. We prove the correctness of updation at an internal node inductively. We assume that the updation is correct at an internal node n_i and then show that the updation is correct according to the description given in Section 3.4.2 in $parent(n_i)$. We prove the correctness of the updation for **Case(1)** in Section 3.4.2, i.e., when n_i is the left child of its parent. When n_i is the right child of its parent, the correctness can be proved in an analogous way. We prove this correctness following the case analysis in Section 3.4.2.

(a) $L\_high(parent(n_i))$

There are two possibilities. If the fields $LL\_cross(n_i)$ and $RR\_cross(n_i)$ contains the same edge e_j, then this edge is free in both the left and right subtrees of n_i and hence e_j is a candidate for $L\_high(parent(n_i))$. However, since we always store the highest x coordinate edge in $L\_high(parent(n_i))$, we have to choose the highest x coordinate edge among $L\_high(n_i)$, $R\_high(n_i)$ and the edge e_j described above.

In the case when the two fields $LL\_cross(n_i)$ and $RR\_cross(n_i)$ do not store the same interval, there are several possibilities. In the first case, both of these fields are empty. In the second case, either the right end point of $LL\_cross(n_i)$ or the left end point of $RR\_cross(n_i)$ extends beyond the subtree rooted at n_i. In this case, we need not consider such an edge for a candidate for $L\_high(parent(n_i))$ since it is not contained in the left subtree of $parent(n_i)$. So, in this case the only candidates for $L\_high(parent(n_i))$ are the intervals $L\_high(n_i)$ and $R\_high(n_i)$.

(c) $LL\_cross(parent(n_i))$

In the case of $LL\_cross(parent(n_i))$, there are two possibilities. Note that, the only two edges in the fields of n_i whose right end point may extend beyond $max(n_i)$ are $LL\_cross(n_i)$ and $RL\_cross(n_i)$. So, $LL\_cross(parent(n_i))$ will be the edge with higher x coordinate among these two edges. If the edge in $LL\_cross(n_i)$ does not extend beyond $max(n_i)$, then the only candidate for this field is $RL\_cross(n_i)$.

(d) $LR\_cross(n_i)$

In this case also, the only fields in n_i whose left end points may extend beyond $min(n_i)$ are $RR\_cross(n_i)$ and $LR\_cross(n_i)$. So, the correctness of the updation is similar to the previous case.

(h) $min(n_i)$

It is clear that $min(parent(n_i)$ is always $min(n_i)$ (since n_i is the left child of its parent) and hence it should be updated if $min(n_i)$ has changed due to the deletion.

(b),(e),(f),(g) All these fields remain unchanged since these are contributed by the right subtree of $parent(n_i)$.

□

3.5 Determination of freedom of the edges

In our algorithm, we need freedom information for edges in several different cases. We first consider how to get the complete freedom information at a particular instance of the execution of the algorithm. In other words, we want to find all the edges which are unblocked in the eastern direction at a particular time of the execution of the algorithm.

Lemma 3.2 If two edges e_i and e_j ($r(e_i) < l(e_j)$) are unblocked in the eastern direction, in $O(\log n)$ time we can find a free edge e_k (if it exists) such that $r(e_i) < l(e_k) < r(e_k) < l(e_j)$.

Proof: We first consider the path from the leaf containing $r(e_i)$ to the root. We call this path as $\mathcal{P}$. We store the possible candidate for the edge e_k in a variable $free\_edge$. While traversing $\mathcal{P}$, at every internal node $n_i \in \mathcal{P}$ (including $root(T)$), we do the following computation. There are two different cases, depending on whether the path $\mathcal{P}$ has come upto n_i from the left or the right subtree of n_i. We first consider the case when $\mathcal{P}$ has come to n_i from its left subtree. We first check whether the edge currently present in *free_edge* is blocked in the eastern direction by the edges present in the fields at n_i. In case $free\_edge$ is blocked, we assign *null* to $free\_edge$. Suppose, n_i has an edge e_m which has the highest x coordinate among the following fields, **(i)** $L\_high(n_i)$, **(ii)** in $LL\_cross(n_i)$ and $RR\_cross(n_i)$ (i.e., both these fields contain the same edge) and **(iii)** $R\_high(n_i)$, and such that e_m is unblocked and completely contained in the subtree rooted at n_i. Similarly, assume that $free\_edge$ currently contains an edge e_n which is unblocked and completely contained in the subtree rooted at $left\_son(n_i)$ as well as in the subtree rooted at n_i. In this case, if e_m is different from e_j, we delete the edge e_n from $free\_edge$ and include

e_m instead. The reason is that if e_m is blocked by some other larger edge e_l, the end points of e_l are outside the subtree rooted at n_i. In that case, e_l will also block the edge e_n in the eastern direction. Hence, if e_n is a free edge, e_m has to be a free edge in the eastern direction. If e_m is same as e_j, we keep $free\_edge$ unchanged.

In the second case, the path $\mathcal{P}$ comes upto n_i through the right child of n_i, we check whether $free\_edge$ is blocked by the edges in the fields in n_i and assign $null$ to $free\_edge$ if it is blocked. We do not assign any new edge in the variable $free\_edge$ since any edge partly or completely present in the subtree rooted at $left\_son(n_i)$ is either blocked by e_i or to the left of $l(e_i)$. This is because the path $\mathcal{P}$ starts at $r(e_i)$.

We claim that if the variable $free\_edge$ contains an edge after we reach $root(T)$ along $\mathcal{P}$, this edge e_k is unblocked in the eastern direction and e_k satisfies the condition in the statement of the lemma. Suppose e_k is not unblocked. In that case, there is some other edge e_l which partly or completely blocks e_k in the eastern direction. In case e_l completely blocks e_k, $r(e_l) > r(e_k)$. Since e_l has a higher x coordinate, it must be present at least at the root or in some node on $\mathcal{P}$ below $root(T)$ as one of the fields. Hence, $free\_edge$ will be set to $null$ after encountering such a field which contains e_l. If e_l partially blocks e_k, e_k cannot be picked up in $free\_edge$ during the traversal of $\mathcal{P}$. In this case, e_k can be present neither as R_high nor as LL_cross and RR_cross (simultaneously) in any internal node along $\mathcal{P}$.

The complete computation takes $O(\log n)$ time, since the length of $\mathcal{P}$ is $O(\log n)$ and we do a constant number of comparisons at every node along $\mathcal{P}$. $\square$

Corollary 3.3 We can find all edges which are free in the eastern direction at a particular stage of the E_tree in $O(k \log n)$ time, where k is the number of vertical edges currently free in the eastern direction.

Proof: Initially, at most three edges may be free in the root of E_tree. These are the edges present in the fields $L\_high(root)$, $R\_high(root)$ and in the fields $LL\_cross(root)$ and $RR\_cross(root)$ (if these two fields contain the same edge). Suppose, we have two free edges e_i and e_j initially, such that $l(e_i) < r(e_i) < l(e_j) < r(e_j)$. We search with the end point $l(e_i)$

to see whether there is a free edge e_k such that $r(e_k) < l(e_i)$. This can be done by a method exactly similar to that in Lemma 3.2 in $O(\log n)$ time. Similarly, any free edge in between e_i and e_j can be found in $O(\log n)$ time. For every such new free edge found, we search again with their end points. Once we have searched with one end point, we do not search again with it. So, for every end point of a newly found free edge, we spend $O(\log n)$ time and hence within $O(k \log n)$ time we can find all the k free edges. $\square$

We now discuss how to update the freedom information when an edge e_i is deleted. We refer to an edge which has got freedom due to the deletion of e_i as an *unblocked edge*. The unblocked edges fall into several categories.

1. An unblocked edge e_j has a partial overlap with the deleted edge e_i.
2. An unblocked edge e_j completely covers the deleted edge e_i.
3. An unblocked edge e_j is completely covered by the deleted edge e_i.

Notice that, after the deletion of an edge e_i, it is sufficient if we find out just one edge e_j which is freed due to the deletion of e_i. After we find such an edge e_j, we can apply the method of Lemma 3.2 to detect other edges which are freed due to the deletion of e_i.

Lemma 3.4 If an edge e_i is deleted from the eastern tree, in $O(\log n)$ time we can find at least one edge (if it exists) which is freed due to the deletion of e_i.

Proof: We prove the lemma for the three cases mentioned above. In all the three cases we first delete the edge e_i (i.e., the leaves containing the end points) and update the fields in all the internal nodes along the deletion paths. After this, we again traverse the deletion paths from the leaf to the root.

First we consider the case when the deleted edge e_i has partial overlap with the unblocked edge e_j. We assume w.l.o.g. that $l(e_i) < l(e_j) < r(e_i) < r(e_j)$ i.e., the left end point of e_j falls inside e_i. In the following, by $lca(n_i, n_j)$ we mean the least common ancestor of two nodes n_i and n_j. Suppose, the leaves containing $r(e_i)$ and $r(e_j)$ are n_i and n_j respectively. Suppose n_k is the node $lca(n_i, n_j)$. n_i and n_j are respectively in the left and right subtree of n_k. If e_j is free in the right subtree of n_k, it will be present in the field $RR\_cross(n_k)$, since the fields in the internal nodes have been already updated after the deletion of e_i. Similarly, if e_j is free in the left subtree of n_k, it will be present in the field $LL\_cross(n_k)$. While traversing the path from n_i to root, we check at every internal node n_l along this path whether $LL\_cross(n_l)$ and $RR\_cross(n_l)$ stores the same edge or not. If we find such an edge, we store this as a candidate for e_j. Note that, e_j cannot be blocked by some edge e_m either completely contained in e_j or having partial overlap with e_j. Since in both of these cases, at least one end point of e_m will be present in the subtree rooted at $lca(n_i, n_j)$ and hence e_j cannot be present in the LL_cross and RR_cross fields of $lca(n_i, n_j)$. It may happen that such an edge e_j is blocked by some larger edge e_n whose end points are outside the subtree rooted at $lca(n_i, n_j)$. In this case, e_n will be present in the fields of some node along the path from $lca(n_i, n_j)$ to $root$. We can easily check this while traversing the path from n_i to $root$.

We now consider the case when the deleted edge e_i is completely covered by an edge e_j which has been freed due to the deletion of e_i. This case is similar to the previous case. If we traverse one of the deletion paths, we will encounter such an edge e_j in an internal node n_k in both the fields $LL\_cross(n_k)$ and $RR\_cross(n_k)$.

In the third case, the deleted edge e_i completely covers an edge e_j which has been freed due to the deletion of e_i. Note that, many small edges may get unblocked due to the deletion of e_i. If we can find at least one such edge, we can find the rest of the unblocked edges by repeated application of the method discussed in Lemma 3.2. So, in the following, we discuss how to find one such unblocked edge e_j.

We traverse a path $\mathcal{P}$ from the leaf containing $l(e_i)$ upto the root. Since e_j is completely contained in e_i, there exists a node n_k on $\mathcal{P}$ with the following property. $\mathcal{P}$ reaches n_k from $left\_son(n_k)$ and e_j is completely or partially contained in the subtree rooted at $right\_son(n_k)$. If e_j is really free after the deletion of e_i, e_j must be present either in $R\_high(n_k)$ or in $RL\_cross(n_k)$. This follows from the definition of R_high and RL_cross fields and the correctness of the updations discussed in Lemma 3.1. Hence at every internal node along $\mathcal{P}$, we check for such an edge e_j according to the above observation. There is a possibility that such an edge is blocked by some other edge e_l. Note that, such an edge e_l cannot have a partial overlap with e_j, since in that case e_j cannot be present in $R\_high(n_k)$ or $RL\_cross(n_k)$. This is because, one end point of e_l will be present in the subtree rooted at $right\_son(n_k)$. Hence, if such an edge e_l exists, it will have both its end points outside the subtree rooted at $right\_son(n_k)$. Hence, we will encounter e_l in the part of $\mathcal{P}$ from n_k upto $root(T)$. e_l will be present in one of the fields of a node in this part of $\mathcal{P}$. We can check this while we continue traversing $\mathcal{P}$. At the end of traversal of $\mathcal{P}$, if such an unblocked edge e_j exists, we can find it. $\square$

4 The Algorithm

We now describe our algorithm for deciding one separability and computing a translational sequence if the input scene is separable in two dimensions. Our algorithm is based on the data structure discussed in the previous section. We elaborate on the informal description of the algorithm given in Section 2.

As mentioned earlier, the basic strategy of our algorithm is similar to that in Chazelle *et al* [1]. We keep four balanced binary trees for the four directions, i.e., *east*, *west*, *north* and *south* and call them as the *E_tree*, *W_tree*, *N_tree* and *S_tree* respectively. Since the scene is completely known, we first sort the vertical (resp. horizontal) edges of the polygons and according to increasing and decreasing x (resp. y) coordinates respectively. We store the vertical edges of the polygons in the E_tree and W_tree according to decreasing and increasing x coordinate respectively. Similarly, the horizontal edges are stored in the N_tree and S_tree according to decreasing and increasing y coordinates respectively. These four trees can be constructed in $O(n \log n)$ time as described in Section 3. After construction of these four trees, we attempt to peel these trees for checking freedom of the edges and the polygons in one of the four directions. In other words, if we are peeling the eastern tree, we are checking the freedom of the edges in the E_tree in the eastern $(+x)$ direction. This is Step 4 in the informal description of the algorithm given in Section 2. We mainly elaborate on this step in this section.

We keep four *free sets* corresponding to each direction. The members of the $Eastern\_free\_set$ or EFS are the vertical edges which are currently free in the eastern direction. Similarly, the other three free sets WFS, NFS and SFS store the edges which are currently free in the western, northern and southern directions respectively. We keep four freedom counters with each polygon P_i. Each counter keeps the count of free edges of P_i in a particular direction. The freedom counter $east\_free\_count_i$ or EFC_i stores the number of vertical edges of polygon P_i which are free in the eastern direction. If the value stored in EFC_i becomes equal to the number of vertical edges of the polygon P_i, then P_i can be translated to infinity in the eastern direction. Similarly, we keep the other three freedom counters NFC_i, WFC_i and SFC_i for each polygon P_i. We now discuss the peeling of these four trees. We assume w.l.o.g. that we start with the eastern tree. Initially, at most three edges may be free at $root(E\_tree)$. These are the edges present in the fields L_high, LL_cross and RR_cross (if these two fields contain the same edge) and R_high. These are the starting members of EFS. After this, we can find all the free edges initially present in E_tree in time proportional to their number multiplied by $O(\log n)$ by the method in Lemma 3.2 and Corollary 3.3. We include all these edges in EFS and delete them from E_tree. Some new edges may become unblocked due to these deletions and we find these new edges by the method described in Lemma 3.4. We repeatedly apply the methods in Corollary 3.3 and Lemma 3.4 until no more new edges are free. While deleting a vertical edge belonging to polygon P_j, we increase EFC_j by one. If for some polygon P_k, EFC_k is equal to the number of its vertical edges, P_k is completely free in the eastern direction. We delete all the vertical edges of P_k from the W_tree and all the horizontal edges of P_k from the N_tree and the S_tree.

When the set EFS cannot be updated anymore, i.e., there is no more free edges (and hence no more free polygons) in the eastern direction, we go to the other three trees in a round robin fashion and try to delete edges from them by the method just described. In other words, if we cannot update the free set in a particular tree, we go to the next tree and so on. If at any stage of the algorithm, we cannot update all the four free sets, the scene is not one-separable. Otherwise, at the end of the algorithm, all the trees become empty and we can derive a separation order from the history of the execution of the algorithm.

Theorem 4.1 The two-dimensional one-separability of a scene of isothetic polygons can be decided in $\Theta(n \log n)$ time and $\Theta(n)$ space, where n is the total number of vertices of the polygons in the scene. A translational ordering of the polygons can also be derived within the same time and space bounds.

Proof: The correctness of the algorithm follows from the description above. If the scene is separable, we are never stuck with the peeling process and the trees become empty at the end. Sorting of the edges and the construction of the four trees take $O(n \log n)$ time. We execute the method of Lemma 3.2 and Corollary 3.3 only twice for each edge in the scene and each time we spend $O(\log n)$ time. Also, each deletion takes $O(\log n)$ time and we spend $O(\log n)$ extra time for a deleted edge to find another edge (if it exists) which is unblocked due to the deletion. This follows from Lemma 3.4. So, the time spent for an edge during the entire execution of the algorithm is $O(\log n)$. Hence, the total execution time is $O(n \log n)$. The space requirement is $O(n)$ since each edge is stored in four leaves and each internal node in a tree stores only a constant number of fields. Also the four freedom counters EFC_i, WFC_i, NFC_i and SFC_i for each polygon P_i takes $O(1)$ space.

The lower bound on space is obvious. The lower bound on the time complexity can be proved from the fact that the *iso-separation* problem (i.e., separating in a single axis-parallel direction) is a special case of the one-separation problem. The iso-separation problem is equivalent to detection of intersection of rectangles. This problem is in turn equivalent to the *element uniqueness* problem which has a lower bound of $\Omega(n \log n)$ [10]. $\square$

5 Extensions to Higher Dimensions

In this section, we discuss the extension of our algorithm to higher dimensions. It is assumed that the reader is familiar with the work of Overmars and Yap [9], since we will use their data structure extensively in this section.

We first discuss the one-separability problem in 3 dimensions and we will indicate briefly how this problem can be solved in higher dimensions with some modifications of the three dimensional solution. Our algorithm is similar to that in [3], however, we achieve the improvement in complexity due to the use of a different data structure. In 3 dimensions, the basic idea of the algorithm is the same as in 2 dimensions. Note that, now there are six orthogonal directions of separation. We keep six separate data structures for these six directions and check in a round robin fashion whether any polyhedron can be translated to infinity along these six directions. However, the data structures employed for this purpose are much more complicated compared to that for the algorithm in two dimensions.

5.1 One-separation in three dimensions

Similar to the case of one-separation in two-dimensions, we keep six data structures in the six orthogonal directions. These six data structures help us to check freedom of the polyhedra in these six directions. In the following, we only discuss the checking of freedom in the $+z$ direction. The checking of freedom in the other five directions is similar.

We construct the data structure in the $+z$ direction in the following way. We first sort all the facets of the polyhedra parallel to the xy plane according to decreasing z coordinate. We sweep a plane parallel to the xy plane from the higher z to the lower z direction. The sweep-plane stops at every facet of the collection of polyhedra in the sorted order. Note that, each position of the sweep plane gives rise to a xy cross-section of the collection of polyhedra. This xy cross-section is a set of (possibly disjoint) polygons parallel to the xy-plane. Since maintaining a set of polygons is difficult, we do the following. We partition the collection of polygons into a set of rectangles. For a n-vertex polygon, this can be easily done in $O(n \log n)$ time and $O(n)$ space. Note that, we are not interested in a minimum partition. For our purpose, any non-overlapping partition into rectangles will do. Each such rectangle in the partition is inserted into the data structure discussed in the next subsection. After the sweep-plane has crossed the minimum z coordinate facet of the collection of polyhedra, our data structure is ready.

5.1.1 The data structure

Our primary data structure is the so-called *orthogonal partition tree* first introduced by Overmars and Yap [9]. This data structure is used to maintain a two dimensional cross section during a space sweep of a collection of three dimensional orthogonal objects. Note that, there may be $O(n^2)$ rectangular regions in such a two dimensional cross section. This is due to the presence of *trellises*. In a trellis, there are two sets of $O(n)$ rectangles each. Each rectangle in the first set intersects all the rectangles in the second set and vice versa. However, Overmars and Yap [9] observed that all the rectangular regions in such a two dimensional cross section need not be represented explicitly. Instead of maintaining the $O(n^2)$ rectangular regions in a trellis, it is sufficient to maintain the two sets of rectangles which give rise to the trellis (at the boundary of the box containing the trellis).

Now, we discuss a two-dimensional version of the orthogonal partition tree. From now on, we use the shorthand OPT_{+z} for this data structure. We use this data structure to store the two dimensional cross sections when we are sweeping the collection of polyhedra from higher to lower z coordinate by a plane parallel to the xy plane. This data structure is used in our algorithm to get information about the freedom of the polyhedra in the $+z$ direction. The other five OPTs in the other five orthogonal directions are constructed in a similar way. OPT_{+z} is a binary tree representing the two-dimensional space in the following way

1. The root of OPT_{+z} represents the whole two-dimensional space.
2. Each internal node $u \in OPT_{+z}$ denotes a (possibly unbounded) rectangular region $\mathcal{R}(u)$. Suppose, the left and right children of u are v and w respectively. Then the regions $\mathcal{R}(v)$ and $\mathcal{R}(w)$ satisfy the following constraints
   - $\mathcal{R}(u) = \mathcal{R}(v) \cup \mathcal{R}(w)$,
   - $\mathcal{R}(v) \cap \mathcal{R}(w) = \emptyset$.
3. The cell $\mathcal{R}(u)$ of each leaf $u \in OPT_{+z}$ is a rectangular cell.

We omit the details of the construction of OPT_{+z} and instead refer to [9]. This construction has been discussed in details by Devine and

Wood [3] as well. First the x axis is partitioned into $\sqrt{n}$ *vertical strips* such that each strip contains at most $2\sqrt{n}$ x coordinates of the sides of the rectangles. Let s be a vertical strip. Let V_s^1 be the set of rectangles in s which have a boundary parallel to the y axis inside s. Note that, $|V_s^1| = O(\sqrt{n})$. Let V_s^2 be the set of rectangles which have a boundary parallel to the x axis inside s. $|V_s^2|$ may be as large as $O(n)$. The strip s is partitioned in the following way. We draw a horizontal line through each boundary (which is parallel to the x-axis) of V_s^1. Moreover, we draw a horizontal line through each $\sqrt{n}$-th boundary of the rectangles in V_s^2. These horizontal lines determine the boundaries of the cells. They also ensure that there is no vertex of the rectangles inside a cell. These cells are the leaves of OPT_{+z}. The internal nodes of OPT_{+z} are formed by first merging the neighboring cells inside a strip and then merging the neighboring strips. This merging process always satisfy the rules discussed above. When OPT_{+z} is constructed in this way, the following properties are true.

Lemma 5.1 *[9]* The orthogonal partition tree OPT_{+z} for n rectangles has the following properties

1. There are $O(n)$ nodes in OPT_{+z},
2. The height of OPT_{+z} is $O(\log n)$,
3. Each rectangle appears in $O(\sqrt{n})$ leaves,
4. No cell contains any vertices of the rectangles,
5. Each leaf or cell contains (parts of) $O(\sqrt{n})$ rectangles.

An interesting property of OPT_{+z} is that no rectangle is stored in the internal nodes and only the leaf nodes store the rectangles. By Property 4 in Lemma 5.1, any rectangle stored in a leaf traverses the associated cell completely. We use some notations from Devine and Wood [3]. The portion of rectangle r that is inside the cell $\mathcal{R}(u)$ of leaf u is called a *fragment* of r. In the following, we call a fragment of a rectangle r as an *x-slab* (resp. *y-slab*) for the cell $\mathcal{R}(u)$, if r has at least one of its vertical (resp. horizontal) boundaries inside $\mathcal{R}(u)$. For a rectangle r and a leaf u, if $r \cap \mathcal{R}(u) \neq \emptyset$, there are three possibilities.

1. r contains $\mathcal{R}(u)$ completely but only partially covers $parent(u)$.

2. $r \cap \mathcal{R}(u)$ is an x-slab in $\mathcal{R}(u)$.

3. $r \cap \mathcal{R}(u)$ is an y-slab in $\mathcal{R}(u)$.

For these three cases, we store r in a secondary data structure associated with $\mathcal{R}(u)$. This secondary data structure is the one used in Section 3, for solving the two-dimensional one-separability problem. For ease of description, we refer to the data structure in Section 3 as a *separation tree.* We keep two separation trees associated with the cell $\mathcal{R}(u)$ of a leaf u. The x-separation tree (resp. y-separation tree) stores the horizontal parts of the *x-slabs* (resp. vertical parts of the *y-slabs*) of the rectangles which satisfy one of the cases mentioned above. Notice that, the x and y separation trees do not represent disjoint space and hence, it is not straightforward to find freedom information from each of the trees alone. In other words, an edge which is part of a rectangle r_i in the x-separation tree and free in this tree, may be blocked by some part of some other rectangle r_2 in the y-separation tree. This happens when r_2 overlaps r_i and the z coordinate of r_2 is higher than the z coordinate of r_1. To avoid this difficulty, we do the following. In addition to the *x-slabs*, we store the horizontal parts of the *y-slabs* which falls within a cell $\mathcal{R}(u)$, in the x-separation tree of $\mathcal{R}(u)$. Similarly, in addition to the *y-slabs*, we store the vertical parts of the *x-slabs* which fall within $\mathcal{R}(u)$ in the y-separation tree of $\mathcal{R}(u)$. When a rectangle r_i completely covers $\mathcal{R}(u)$, the horizontal and vertical extents of r_i are stored in the x and y separation trees respectively. However, these two separation trees have an important difference from the data structure in Section 3. Note that, we are using these two trees to detect freedom of edges in the z direction, whereas in Section 3, we were checking freedom either in the x or in the y direction. However, we can easily modify the data structure for our purpose. Note that, during the construction and maintenance of the data structure in Section 3, the main operation is the function $High(.)$. Here $High$ returns the highest x coordinate edge among its arguments if we are checking freedom in the $+x$ direction. In case of our x and y separation trees, we use the function $High$ to always return the highest z coordinate edge among its arguments.

Note that, even if we detect that a fragment f_i is free inside a a cell c_j, it may be blocked by some part of a large rectangle which is not stored in c_j. For example consider a large rectangle r_k which covers the complete domain of the collection of rectangles and which has the highest z coordinate. According to the construction of OPT_{+z}, such a rectangle will be stored only in the cells which lie on the boundary of the domain. However, any rectangle whose fragments are stored in other cells and which has a lower z coordinate will be blocked by r_k in the $+z$ direction. We introduce an additional list $cover(u)$ for every internal node $u \in OPT_{+z}$ to detect such blockings. A rectangle which completely covers the region denoted by $u \in OPT_{+z}$, but which only partially covers the region of $parent(u)$ is a member of $cover(u)$. We keep the members of $cover(u)$ for every internal node u in the descending order of z coordinates. This is easy to keep when we are sweeping from higher to lower z coordinate during the construction of OPT_{+z}. Note that, a particular rectangle can appear in at most $O(\log n)$ cover lists.

While doing the space sweep from higher to lower z direction, inserting a particular rectangle r_i in OPT_{+z} requires the determination of the leaves in OPT_{+z} whose regions intersect r_i. Property 3 of Lemma 5.1 ensures that $O(\sqrt{n})$ time is required to identify the fragments of r_i. By Property 5, each leaf holds $O(\sqrt{n})$ fragments and therefore the height of the two separation trees associated with each leaf is $O(\log \sqrt{n}) = O(\log n)$.

5.1.2 Detecting freedom in the $+z$ direction

In this subsection, we briefly discuss how to determine the freedom of a particular polyhedron in the $+z$ direction. As mentioned earlier, all our discussion is valid with slight modifications for the other five orthogonal directions.

During the construction of the data structure (i.e., during the space sweep), each polyhedron facet is divided into a set of rectangles. Note that, for the polyhedron P_i, if all the rectangles corresponding to all of its facets parallel to the xy plane are free in the $+z$ direction, we can move P_i to infinity in the $+z$ direction. Hence, for a rectangle r_i in the decomposition, we tag r_i with the identitiy of the polyhedron P_j of which r_i is a part. The total number of rectangles which a polyhedron P_j gives rise to, is called the *rectangle count* of P_j or $RC(P_j)$. The number of rectangles which originate from P_j and which are free in the $+z$ direction, is called the *free rectangle count* of P_j or $FRC(P_j)$. Similarly, a rectangle r_i may get divided in $O(\sqrt{n})$ leaves of OPT_{+z}. These are the fragments of r_i. The total number of fragments of a rectangle r_i is called the *fragment count* of r_i or $FC(r_i)$. The number of fragments which originate from the rectangle r_i and which are free in the $+z$ direction is called the *free fragment count* of r_i or $FFC(r_i)$. The rectangle r_i is free in the $+z$ direction when all of its fragments are free in the $+z$ direction. Consider the j-th fragment $f_j(r_i)$ of the rectangle r_i. If all four sides of $f_j(r_i)$ are free in the $+z$ direction, then $f_j(r_i)$ is also free in the $+z$ direction. We determine the freedom of a particular polyhedron P_j in the following way. We check whether $RC(P_j) = FRC(P_j)$. To check this, we have to first check whether $FC(r_i) = FFC(r_i)$ for each rectangle r_i which contributes to $RC(P_j)$. In the rest of this subsection, we elaborate briefly on this computation.

Initially, we visit all the leaves of OPT_{+z} and update $FFC(r_i)$ for all the rectangles r_i whose fragments are free. This is done by examining the x and y separation trees of the leaves. Note that, if an edge is free in these two separation trees, we have to still check its freedom against the *cover* lists stored in the internal nodes of OPT_{+z}. This can be done easily in additional $O(\log n)$ time by going up from the leaf to the root of OPT_{+z}. We need to check the z coordinate of an edge only against the top-most element of the cover list of a node. We delete the edges of the free fragments and update both the x and y separation trees of a leaf of OPT_{+z} as a result of these deletions. There may be new free edges due to these deletions. These edges can be found by the method in Lemma 3.2 and Lemma 3.4. Whenever we delete a fragment, we increase the FFC of the corresponding rectangle. If the FFC of a particular rectangle becomes equal to the FC of the same rectangle, we increase the FRC count of the corresponding polyhedron. We also delete such a rectangle from all the cover lists of the internal nodes of OPT_{+z} where this rectangle appears. This takes $O(\log n)$ time. Notice that, when we are detecting freedom in $+z$ direction, only the top-most elements of the cover lists will get deleted. However, when we are detecting freedom in one of the other orthogonal directions and deleting a polyhedron from OPT_{+z}, any member of the cover set may get deleted. But it is easy to keep track of the rectangles in the cover lists such that a single such deletion can be performed in $O(1)$ time. The details are omitted. If for a particular polyhedron P_j, $RC(P_j)$ becomes equal to $FRC(P_j)$, we delete all the rectangles and their fragments which originate from P_j. This deletion is done in all the x and y separation trees of OPT_{+z} where the fragments of P_j exists. Also, we delete the rectangles of P_j from all the other five orthogonal partition trees in the five orthogonal directions. This deletion can be done in time proportional to the number of fragments since we have kept track of all the fragments of P_j during the construction of OPT_{+z} and the other OPTs.

5.1.3 The algorithm and its complexity

Our algorithm is similar to that for the two-dimensional one-separation problem. We visit each of the six orthogonal partition trees (OPTs) in a round robin fashion. If no more rectangles or polyhedrons can be freed from a particular OPT, we go to the next OPT. If a polyhedron P_j is freed in a particular OPT (i.e., in a particular direction), we delete all the rectangles and fragments of P_j from all other OPTs. If the scene is one-separable, at the end of this computation all the OPTs will be empty. Otherwise, sometime during this process, we cannot separate any polyhedron from any of the OPTs. If this is the case, the scene is not one-separable.

Theorem 5.2 The three dimensional one-separation problem can be solved in $O(n\sqrt{n}\log n)$ time and $O(n\sqrt{n}\log n)$ space, where n is the total number of vertices of the polyhedra.

Proof: The initial sorting and space sweep in the six directions take $O(n\log n)$ time. The decomposition of the facets of the polyhedra creates $O(n)$ rectangles. Each rectangle is broken up into $O(\sqrt{n})$ fragments. Inserting a fragment in a separation tree takes $O(\log n)$ time. Hence, the complete construction of the OPTs take $O(n\sqrt{n}\log n)$ time. The analysis of the peeling of the OPTs is similar to that in Section 3. It suffices to say that we spend $O(\log n)$ time for deleting each fragment. Hence, we achieve the overall time complexity of $O(n\sqrt{n}\log n)$. The space complexity is $O(n\sqrt{n}\log n)$ since each rectangle may be present in $O(\log n)$ cover lists. All the separation trees in the leaves of OPT_{+z} together takes

$O(n\sqrt{n})$ space since there are $O(n\sqrt{n})$ fragments. $\square$

5.2 One-separation in d-dimensions

The solution of the d-dimensional problem is similar to that of the three-dimensional problem and hence we only briefly indicate the algorithm. We sweep the d-dimensional space and build a $(d-1)$ dimensional orthogonal partition tree for each of the $2d$ orthogonal directions. Each leaf of such an OPT represents a $(d-1)$ dimensional cell. We construct $(d-1)$ separation trees for each such cell. The rest of the algorithm is similar to the algorithm for three dimensional one-separation. It suffices to say that we again apply the method of checking the $2d$ orthogonal partition trees for freedom of the polyhedra. When we are checking a particular OPT, we check for the freedom of the edges in the $(d-1)$ separation trees at the leaves of this OPT. The complexity of the algorithm follows from the following result from [9].

Lemma 5.3 *[9]* A d-dimensional orthogonal partition tree for n d-dimensional boxes has the following properties.

1. The OPT has $O(n^{d/2})$ nodes,
2. Each d-box is stored in $O(n^{(d-1)/2})$ leaves,
3. There is no vertex in a cell of a leaf and
4. Each leaf holds at most $O(\sqrt{n})$ d-boxes.

Theorem 5.4 The d-dimensional one-separability problem can be solved in $O(dn^{d/2}\log n)$ time and $O(dn^{d/2}\log n)$ space, where n is the total number of vertices of the polyhedra.

6 Conclusion

We have presented the first optimal algorithm for the one-separation problem in the plane. This algorithm achieves optimality with respect to both time and space. We have also introduced a novel data structure which may find other applications. In d-dimensions, our algorithms are better than the best existing algorithms by a factor of $O(\log^2 n)$ in the time complexity. However, several interesting problems remain open. In the plane, instead of allowing a single move in a single orthogonal direction, it is more practical to allow multiple moves. Is there an efficient algorithm to solve the k-separability problem for $k > 1$, where k is the number of allowed translations for a single polygon? In higher dimensions, we have been unable to utilize the *streaming technique* of Overmars and Yap [9] to reduce the space complexities of our algorithm. It is an interesting open problem whether the complexities presented in this paper can be improved even in 3 dimensions.

References

[1] B. Chazelle, Th. Ottmann, E. Soisalon-Soininen and D. Wood. "The complexity and decidability of $SEPARATION^{TM}$" , *Proc. International Colloquium on Automata, Languages and Programming*, (1984), *LNCS* **172**, pp. 119-127.

[2] F. Dehne and J. -R. Sack. "Translation separability of sets of polygons", *The Visual Computer*, **3**, (1987), pp. 227-235.

[3] M. Devine and D. Wood. "$SEPARATION^{TM}$ in d dimensions or strip mining in asteroid fields", *Comput. and Graphics*, Vol. **13**, No. **3**, (1989), pp. 329-336.

[4] L. J. Guibas and F. F. Yao. "On translating a set of rectangles", in *Advances in Computing Research, Volume I: Computational Geometry*, Ed. F. P. Preparata, JAI Press Inc., (1983), pp. 61-77.

[5] B. K. Natarajan. "On planning assemblies", *Proc. Fourth Annual ACM Symp. on Computational Geomtry*, (1988), pp. 299-308.

[6] O. Nurmi. "On translating a set of objects in 2- and 3- dimensional space", *Computer Vision, Graphics and Image Processing*, Vol. **36**, (1986), pp. 42-52.

[7] D. Nussbaum and J. -R. Sack. "Disassembling two-dimensional composite parts via translation", *International Journal of Computational Geometry & Applications*, Vol. **3**, No. **1**, (1993), pp. 71-84.

[8] Th. Ottmann and P. Widmayer, "On translating a set of line segments", *Computer Vision, Graphics and Image Processing*, Vol. **24**, (1983), pp. 382-389.

[9] M. H. Overmars and C. K. Yap. "New upper bounds in Klee's measure problem", *SIAM J. Computing*, Vol. **20**, No. **6**, (1991), pp. 1034-1045.

[10] F. P. Preparata and M. I. Shamos. *Computational Geometry : an Introduction*, Springer-Verlag, 1985.

[11] J. Reif. "Complexity of the mover's problem and generalizations", *Proc. 20th IEEE Symposium on Foundations of Computer Science*, (1979), pp. 560-570.

[12] G. T. Toussaint. "Movable separability of sets", in *Computational Geometry*, Ed. G. T. Toussaint, North-Holland, (1985), pp. 335-375.