Theoretical Computer Science 103 (1992) 107–135
Elsevier

Minimum vertex hulls for polyhedral domains*

Gautam Das
Memphis State University, Memphis, TN 38152, USA

Deborah Joseph
University of Wisconsin, Madison, WI 53706, USA

Abstract

Das, G. and D. Joseph, Minimum vertex hulls for polyhedral domains, Theoretical Computer Science 103 (1992) 107–135.

Given a collection of pairwise disjoint polygons on the plane, we wish to cover each polygon with an exterior hull such that (1) the hulls are pairwise disjoint, and (2) the total number of vertices of the hulls is minimized. This problem has applications in the area of object approximations. Various versions of this problem (in two, and higher dimensions) are shown to be NP-hard. The paper also describes several approximations and exact algorithms for the problem.

1. Introduction

In this paper we investigate several variations of the following problem. Suppose we are given a collection of pairwise disjoint polygons on the plane. We are required to cover each with a polygonal hull such that (1) the hulls are pairwise disjoint and (2) the total number of vertices (or equivalently, edges) of the hulls is minimized.

This problem belongs to the general area of *object approximations*, where the goal is to approximate complex objects by simpler shapes [1, 2, 4, 12, 14, 16, 20]. There are many variations possible to this problem. It can be restricted to special types of polygons such as rectilinear polygons or convex polygons. It can also be generalized to three dimensions, where polyhedra replace polygons. Since the surface of a polyhedron is usually described by vertices, edges, and faces, we may be interested in

*This work was accomplished while the first author was a graduate student at the University of Wisconsin, Madison. It was supported in part by the National Science Foundation under NSF PYI grant DCR-8402375. A preliminary version was presented at the Seventh Symposium on Theoretical Aspects of Computer Science, Rouen, France, 1990.

finding hulls that minimize any, or all, of these three quantities. We observe that if the polygons (polyhedra) are sufficiently far apart, minimum hulls can easily be obtained by enclosing each object in a triangle (tetrahedron). However, if the objects are placed closer together, then because we require that the hulls are disjoint, the problem is nontrivial.

Our problem can be applied in many situations. There are a wide range of geometric applications, such as *circuit design*, *robotics*, *motion planning*, etc., where the input is specified as a collection of disjoint polyhedra. In many cases these problems are computationally quite complex to solve. Consequently, it is desirable to investigate them beyond worst-case complexity, and look for algorithms that exploit typical problem instances. Our idea of replacing polyhedra by their minimum hulls, if efficiently accomplished, can be used as a preprocessing step to speed up subsequent processing because of the reduction in input size. During the preprocessing, care must be taken to ensure that the hulls adequately represent the original input, in the sense that the final output is not significantly compromised. More precise criteria will, of course, depend upon the actual problem being considered.

Some examples of these applicatons are as follows. In *circuit design*, rectilinear polygons embedded on a plane may represent circuit components, and the problem is to lay out wires that avoid these obstacles. The objective may be to avoid wire crossing, or reduce wire length. Similarly, in *robot motion planning*, disjoint polyhedra represent obstacles inside a workspace in which the robot has to navigate. The robot itself may be modeled as a polyhedron. Some of the problems involve, computing the region reachable by the robot, or computing shortest routes for the robot's motion [3, 6, 17]. Another application involves path planning between the objects based on the *link metric*, which represents the number of vertices, or *bends* in the path [18]. Apart from reducing the input size, our preprocessing also removes bends in the shapes of the objects.

A problem from combinatorial geometry [16], which is related to our problem and which arose during the study of *stochastic automata*, has been posed by Klee [13]. The problem is: Given two concentric convex polyhedra in three dimensions, fit a minimum (in vertices, edges, or faces) polyhedron that nests between them. We observe that to be minimum, the intermediate polyhedron has to be convex. The original formulation was for minimizing vertices for polytopes in arbitrary dimensions. Polynomial-time algorithms have been found for the two-dimensional problem [1, 12, 20], but no satisfactory solutions exist for higher dimensions.

We first prove that our problems are NP-hard in both two and three dimensions. We then describe some algorithms that solve specific variants of the problem. In more detail, our results are as follows.

We show the two-dimensional minimum-hulls problem to be NP-hard. The proof follows from a reduction from the Planar-3SAT problem [15]. The problem is NP-hard even when restricted to convex polygons, or convex rectilinear polygons. If the number of polygons is *bounded*, we do not believe that the problem is intractable, because our reduction needs an arbitrary number of polygons.

In three dimensions the minimum-hulls problem is also NP-hard (when minimizing vertices, edges, or faces), by trivially extending the two-dimensional proofs. Again, this result holds for convex polyhedra and for convex rectilinear polyhedra.

We next develop algorithms for the special problem of computing minimum rectilinear hulls on the plane. Our first result here is an efficient approximation algorithm which constructs hulls that achieve more than half the maximum possible reduction in vertices. In other words, if n is the number of vertices in the input, and m is the number of vertices in the minimum hulls, then our algorithm computes a set of hulls with at most $\frac{1}{2} \cdot (n+m)$ vertices. This algorithm runs in $O(n \log n)$ time. It has applications in the computation of shortest rectilinear paths amidst polygonal obstacles.

We also design an algorithm for constructing *actual* minimum rectilinear hulls which runs in $O(n^{2k+9})$ time, where k is the number of polygons. The algorithm is based on dynamic programming. The theoretical significance of the algorithm is that the difficulty of the problem is related more to the number of polygons than to their shapes and positions.

Preliminary versions of some of the above results appeared in [7]. In that paper, we also had a nontrivial three-dimensional result (which we do not reproduce here), where we show that the problem of computing minimum hulls for *only two* nonconvex polyhedra is NP-hard. Thus, unlike the two-dimensional case, where the hardness is due to the number of polygons, here the difficulty lies in the complex shapes that three-dimensional objects can have. Very recently, we have shown that Klee's problem of nesting a polyhedron with minimum faces between two concentric convex polyhedra is NP-hard (a preliminary version may be found in [8]). As a corollary, our problem for constructing minimum-faced hulls for only two disjoint polyhedra, of which one is *convex*, is NP-hard. From a theoretical standpoint, this is one of the few known intractable results in three-dimensional computational geometry that involves the *simplest* (and *fewest*) objects, namely three convex polyhedra. These intractability results will formally appear in a future paper.

Our research differs from most previous research works on object approximations in two important ways. First, previous goals of approximations have usually been to minimize continuous measures (for instance, minimizing the *area* of the hull [14], or minimizing the *symmetric difference* between the area of the hull and the object [2]), while ours is combinatorial, as in [1]. Second, complex objects have usually been considered in isolation, while we have a more restrictive environment where a number of neighboring objects can hinder the approximation process.

The rest of the paper is organized as follows. Sections 2 and 3 describe the NP-hardness proofs of our two-dimensional and three-dimensional problems, respectively. Sections 4 and 5 describe the approximation algorithm for constructing rectilinear hulls on the plane, while Section 6 describes the exact algorithm for constructing rectilinear minimum hulls. We conclude with a list of open problems.

2. NP-hardness of two-dimensional minimum hulls

A *polygon* is a piecewise linear simple curve on the plane, where the linear fragments of the boundary are *edges*, and edges meet at points, or *vertices*. A *convex polygon* is one whose enclosed region is convex. A *rectilinear polygon* is one where each edge is either horizontal or vertical. A *convex rectilinear polygon* is a rectilinear polygon such that, if the two endpoints of any vertical (or horizontal) line segment are inside the polygon, then the rest of the line segment is also inside the polygon.

In this section we will show that computing minimum hulls for a set of disjoint polygons on the plane is NP-hard, even when the polygons are convex, or convex rectilinear. We shall first prove it for convex rectilinear polygons, and then apply very similar ideas to prove it for convex polygons.

Theorem 2.1. *Computing minimum rectilinear hulls for convex rectilinear polygons is* NP-*hard.*

Proof. Throughout the proof, a polygon will refer to a convex rectilinear polygon, unless otherwise mentioned. We first introduce some definitions and notations. Let $P=\{p_1, \ldots, p_m\}$ be a set of pairwise disjoint simple polygons on the plane. Let $H=\{h_1, \ldots, h_m\}$ be another such disjoint set of polygons.

Define *Valid-Hull*$(H, P)=$*true* if

(1) h_i contains p_i, for all i,

(2) for any edge e of h_i, there exists a parallel edge f of p_i such that e and f overlap, and *false* otherwise.

Condition (2) is a more complicated way of saying that the hull h_i is *tight*, that is, each edge has been pushed inwards until it encounters an edge of p_i.

Let *MinHull* be a valid (under the above criteria) set of hulls of P with the minimum number of vertices. Note that *MinHull* may not be unique. We will prove that, given P, computing a *MinHull* is NP-hard. Suppose the *Valid-Hull* predicate was redefined by eliminating condition (2), that is, edges of h_i need not touch the boundary of p_i. If we are given a *MinHull* under these relaxed conditions, it is then quite easy to convert it to a tight *MinHull* by pushing each edge towards the interior of the hull until it touches the polygon. In the process we may be lengthening or shortening adjacent edges but otherwise retaining the same shapes of the hulls. Thus, adding condition (2) does not make the computation of *MinHull* polynomially harder. In the proof, *MinHull* will be always tight, unless otherwise stated. Condition (2) has simply been included so as to make the proof easier.

We next prove the following graph-theoretic lemma, which will be useful in the construction of the polynomial reduction. The lemma describes a particular way of drawing a planar graph on a plane.

Lemma 2.2. *Given any planar graph* G (*Fig.* 1), *it may be drawn on a plane in polynomial time such that*

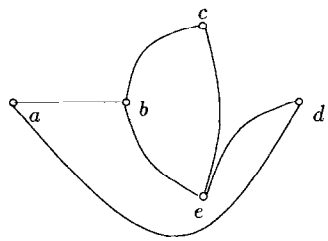


Fig. 1.

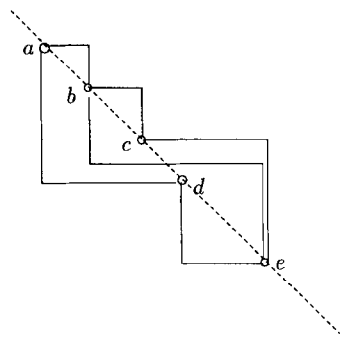


Fig. 2.

(1) *all vertices are aligned on a straight line tilted at some angle with the horizontal, and*

(2) *each edge is a staircase, composed of a polynomial number of alternating vertical and horizontal line segments, where each segment intersects the tilted line (Fig. 2). Thus, if several horizontal (or vertical) line segments are incident at a vertex, we may imagine that they are drawn in parallel, separated by infinitesimal distances.*

Proof of Lemma 2.2. First draw a *Fary embedding* of G (Fig. 3), using the polynomial-time algorithm given in [10]. A Fary embedding is a planar drawing where the edges are straight line segments. It is known that every planar graph can be drawn this way.

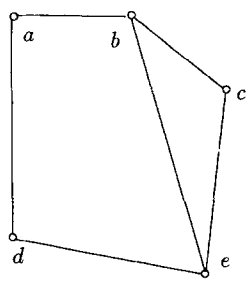


Fig. 3.

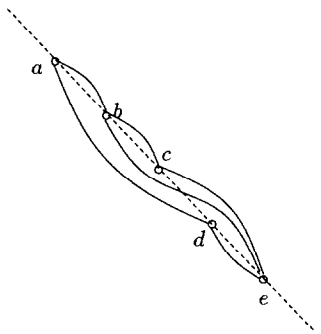


Fig. 4.

We now assume that each edge is a stretched rubber band. Draw a tilted line on the plane, and move each vertex towards the line, in a direction perpendicular to it. We eventually obtain a drawing shown in Fig. 4. Observe that each rubber band wriggles along the tilted line, and the number of wriggles is no more than the number of vertices in the graph. Now each edge can be redrawn by replacing the sequence of wriggles with a sequence of alternating vertical and horizontal line segments, resulting in the drawing in Fig. 2. It is easy to see that the total number of such segments is polynomial in the size of the entire graph. □

Proof of Theorem 2.1 (*conclusion*). We are now ready to prove the theorem. We shall prove that the equivalent decision problem is NP-complete. Let $\#(P)$ denote the number of vertices in a set of polygons P. The decision problem is formally stated as follows. Given a set of pairwise disjoint convex polygons P, and an integer k, is there a valid set of hulls H, such that $\#(P) - \#(H) \geq k$? Clearly, the problem is in NP because a nondeterministically constructed H can be verified in polynomial time. For proving NP-completeness, the reduction shall be from *Planar-3SAT* [15], which we define below.

A *variable–clause* graph of a 3SAT instance (with n variables and m clauses) is defined as a bipartite graph where vertices are variables and clauses, and edges are between variable vertices and clause vertices. If a clause C has a literal of a variable V, then $[C, V]$ is an edge. Thus, the graph has $3m$ edges. Also, the edges are marked + if a positive literal is used, − if a negative literal is used. Planar-3SAT may now be formally stated as follows. Given a planar variable-clause graph for a 3SAT formula, is there a satisfying truth assignment? The reduction will proceed in a series of steps.

Step 1: For the given variable–clause graph, construct a planar drawing as in Lemma 2.2.

In the remaining steps, we shall construct components for variables, clauses, and edges of the planar drawing of Step 1. Each component will in turn be a collection of convex polygons. Finally, all components will be "superimposed" upon the planar

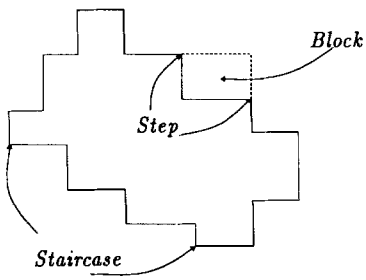


Fig. 5.

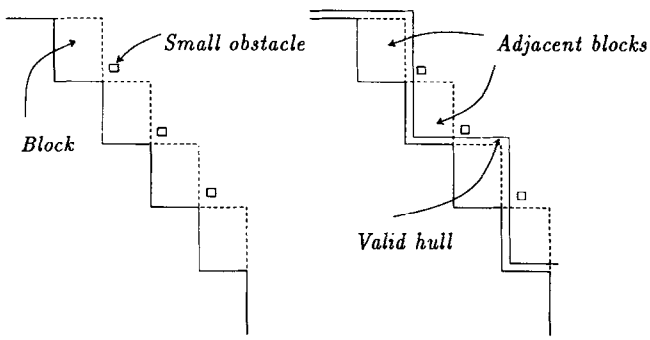


Fig. 6.

drawing of Step 1, so as to have a global collection of convex polygons. The planarity of the drawing will ensure that these polygons remain pairwise disjoint. The following definitions will be useful in describing the remaining steps.

Consider a convex polygon p_i. Its boundary has at most four *staircases*, as shown in Fig. 5. Each staircase is composed of a sequence of adjacent *steps*. Thus, each step consists of a vertical edge and an adjacent horizontal edge. Each step defines a rectangular region outside the polygon called a *block*. A block is *selected* if a hull h_i completely encloses it. Consider a staircase of p_i. Suppose we place small, square obstacles along its length, as shown in Fig. 6. Because a valid hull h_i has to be tight, this will ensure that adjacent blocks cannot be simultaneously selected. In all our constructions we will assume that such small obstacles are placed near each staircase to force the above condition, so they will not be explicitly depicted any further. We observe that a valid hull h_i is basically the union of p_i with possibly some selected blocks of p_i such that no two selected blocks are adjacent. It can be seen that no two blocks belonging to the same polygon intersect. But two blocks belonging to different polygons can, as shown in Fig. 7. Thus, if h_i and h_j are valid hulls of p_i and p_j, and block b is enclosed within h_i, then block c cannot be within h_j. We can see that, given a set of polygons P of the type described above (that is, with small obstacles placed

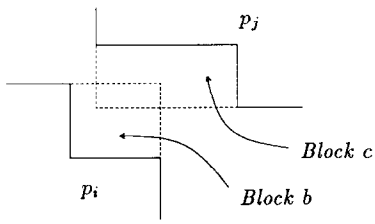


Fig. 7.

near staircases, etc.), a valid set of hulls H is the union of P together with a selection of blocks such that no two selected blocks intersect, or are adjacent. To construct a *MinHull*, it is to our advantage to select as many such blocks as possible.

Another way of expressing this idea is by using graph-theoretic techniques. Define a *block-graph* where (1) blocks are vertices, and (2) $[a, b]$ is an edge if blocks a and b are either adjacent, or intersect with each other. Thus, a valid set H of hulls contains blocks that form an independent set in the block-graph. To obtain *MinHull*, it is sufficient to compute a maximum independent set of the block-graph.

Step 2 (*variable component*): We shall illustrate this component by an example. There is one such component per variable, and each component is dependent in its design on the total number of clauses. Suppose there are 4 clauses in the Planar-3SAT instance. Figure 8 illustrates a variable component. There is a main polygon with two staircases, and a pair of *connecting* polygons at either end. All blocks in this component are classified as *positive*, *negative* or *connecting* (see Fig. 9 for the block-graph). In this example there are four positive blocks and four negative blocks on either staircase of the main polygon. There are totally $6+6+4\cdot 4=28$ blocks in the component. In the general case, if m is the number of clauses, there are $6+6+4m$ blocks. Clearly, as Figs. 10 and 11 indicate, there are two ways of selecting maximum independent sets in the block-graph, one in which all positive vertices are selected and the other in which

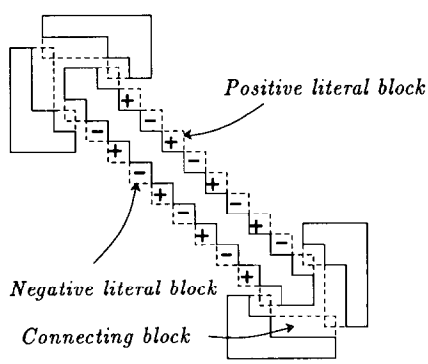


Fig. 8.

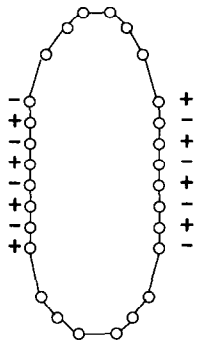
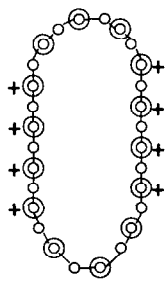


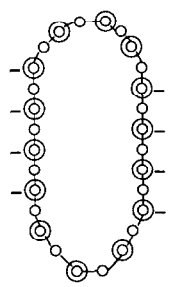
Fig. 9.



Positive blocks selected

Variable set false

Fig. 10.



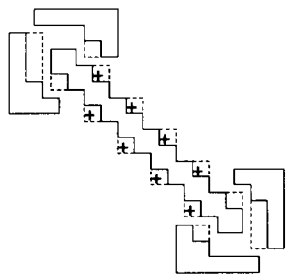
Negative blocks selected

Variable set true

Fig. 11.

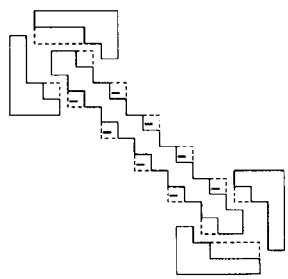
all negative vertices are selected. This also corresponds to the two best ways of designing hulls for this component; see Figs. 12 and 13.

In one case, when the positive blocks are selected, the variable is set *false* and in the other case, *true*. In either case, for this component only, $\#(P) - \#(H) = 2 \cdot 14 = 28$. In



False

Fig. 12.



True

Fig. 13.

the general case, $\#(P)-\#(H)=2(6+2m)$. We should note that we could also have valid hulls where positive and negative blocks are both selected, but in these cases we would not achieve the same reduction in the number of vertices. Finally, we can imagine each variable component as a macro vertex in the planar drawing which was constructed in Step 1.

Step 3 (*clause component*): We shall describe this component by first describing its block-graph; see Fig. 14. There are three *literal* blocks and 24 *connecting* blocks such

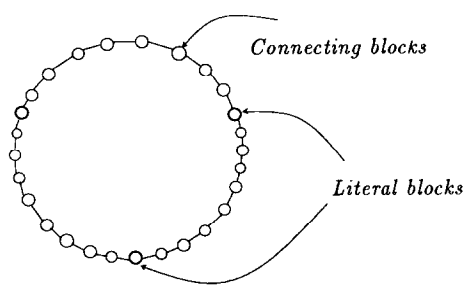
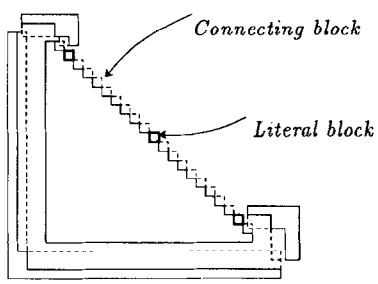


Fig. 14.

that the graph is a cycle, with 8 connecting blocks in between any two literal blocks. By inspection, we can see that any maximum independent set of the block-graph has to select at least one literal block. To realize this block-graph by polygons, we notice that in the planar drawing of Step 1, a clause vertex has either all three incident edges in one half plane, or two in one half plane and the third in the other. These two cases are realized in Figs. 15 and 16.

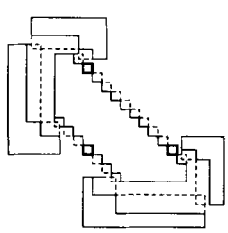
Since the maximum independent set has size 13 in the block-graph, for this component only, $\#(P) - \#(H) = 26$ for the best hulls. We provide some motivation for such a design. Each clause component will be a macrovertex of the planar drawing of Step 1, just like the variable components. These components will be connected via edge components, which will be described later. A clause is satisfied if a maximum independent set can be constructed for its block-graph. This can happen only if we are allowed to select at least one of the literal blocks.

Step 4 (*edge component*): We shall illustrate this component by an example. Consider an edge of the planar drawing of Step 1, as shown in Fig. 17. The edge connects a clause with a variable. The label on the edge indicates that the clause contains a negative literal of the variable. The edge component is realized in Fig. 18, and has



Symmetric cases exist

Fig. 15.



Symmetric cases exist

Fig. 16.

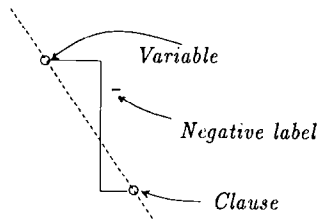


Fig. 17.

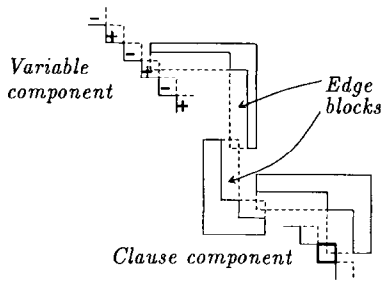


Fig. 18.

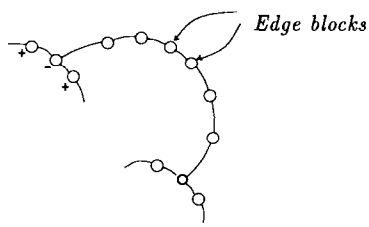


Fig. 19.

altogether 6 blocks. If we consider its block-graph (Fig. 19), the blocks are linked as a chain. The block at the upper (lower) end of the chain is positioned to intersect with a negative literal block of the variable component (literal block of the clause component).

In the general case, an edge component has an even number of blocks linked as a chain, with the end blocks intersecting with appropriate blocks of the variable and clause components. The actual number of blocks is determined by the number of vertical and horizontal line segments that form the edge in the planar drawing. Since an edge component has an even number of blocks, its best hull has to select at least one of the end blocks. In that case, the block that intersects it (which may belong to either the variable or the clause component) will not be selected. Thus, an edge component propagates the *true* or *false* state of a variable component to a clause component.

We can imagine each edge component being laid out along its corresponding edge in the planar drawing of Step 1. Since there are enough positive and negative blocks on either side of the main polygon in any variable component, all edge components may be laid out without encountering polygon crossover situations. We observe that the edge components provide a means of linking up the block-graphs of the variable and clause components. Thus, finding a *MinHull* for the entire set of components is equivalent to finding a maximum independent set for the entire block-graph.

Clearly, the whole reduction requires only polynomial time. Suppose the Planar-3SAT instance had n variables and m clauses. Let P be the set of polygons of all components. Let the total number of blocks of all edge components be r. Let $k=(12+4m)n+26m+r$. Note that the first term corresponds to the reduction in vertices if all variable components had their best hulls. Similarly, the second and third terms correspond to clause and edge components, respectively.

We pose the following problem. Is there an H such that *Valid-Hull*$(H, P)=$*true* and $\#(P)-\#(H)\geq k$? From the reduction, we can conclude that this is true if and only if the instance of Planar-3SAT is satisfiable. Thus, constructing a *MinHull* for a set of convex rectilinear polygons is NP-hard, and the theorem is proved. □

We will now prove that obtaining minimum hulls for convex (nonrectilinear) polygons is also NP-hard. We shall do this by a very similar reduction from Planar-3SAT.

Theorem 2.3. *Computing minimum hulls for convex polygons is* NP*-hard.*

Proof. Consider the decision version of our problem, which may be stated as follows. Given a convex polygon set P, and an integer k, is there a set H of hulls such that $\#(P)-\#(H)\geq k$?

For proving NP-completeness, the reduction will be from Planar-3SAT. We will design components for variables, clauses, and edges of a given planar variable–clause graph of some Planar-3SAT instance. Each component will be a collection of convex polygons. All components will be superimposed on the planar graph, so that we have a global collection of convex polygons. The planarity of the drawing will ensure that these polygons remain pairwise disjoint.

The following definitions will be useful in describing the remaining steps. Consider the two groups of convex polygons shown in Fig. 20. The central quadrilateral of each group defines an empty triangular region called a *block*, which would be filled if the quadrilateral was extended into a triangle. Thus, a minimum hull for the quadrilateral either *selects* the block, or excludes it. Two blocks belonging to different groups can intersect as the figure shows. If the minimum hulls of one group selects its block, clearly, the hulls of the other group will have to exclude the intersecting block. Throughout our construction we place only such groups on the plane so that

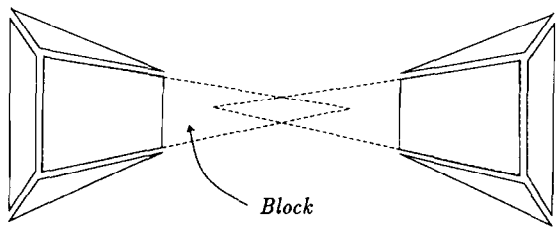


Fig. 20.

appropriate blocks intersect. To construct minimum hulls, it is to our advantage to select as many such blocks as possible.

We are now ready to describe our reduction. It first proceeds exactly as in Theorem 2.1. The reduction of Theorem 2.3 will be complete if we can replace the rectilinear polygons by nonrectilinear polygons. This is done by replacing the rectangle blocks by triangle blocks (defined by polygon groups as described above), such that pairs of adjacent or intersecting rectangle blocks are replaced by pairs of intersecting triangle blocks.

We shall illustrate that this is possible, by an example. Consider Fig. 21, which shows a portion of the final arrangement of rectilinear polygons in the reduction of Theorem 2.1. The corresponding portion of the block-graph is shown in Fig. 22. We

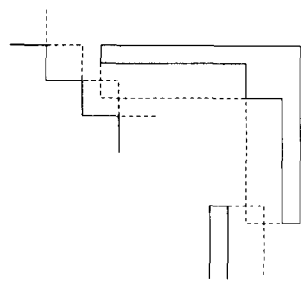


Fig. 21.

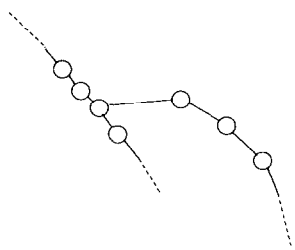


Fig. 22.

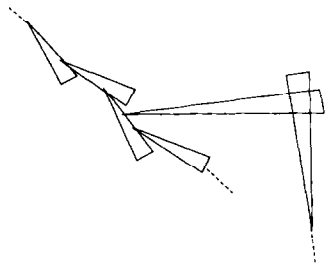


Fig. 23.

replace the rectilinear polygons by nonrectilinear polygon groups whose blocks appropriately intersect, as in Fig. 23. It is easy to see that this kind of replacement can be accomplished for the whole graph.

The reduction clearly shows that the problem of finding minimum hulls for convex polygons is NP-hard. □

3. NP-hardness of three-dimensional minimum hulls

In this section we shall prove NP-hardness for some variants of our problem in three dimensions.

A *polyhedron* is a piecewise linear closed surface in three-dimensional space, where the pieces of the surface are *faces*. Faces meet at *edges* and edges meet at *vertices*. All definitions we have encountered for two-dimensional polygons easily generalize to three-dimensional polyhedra.

How hard is it to compute minimum rectilinear hulls (in number of vertices, edges, or faces) for a set of convex rectilinear polyhedra in space? This is easily seen to be NP-hard, by trivially extending the proof of Theorem 2.1 as follows. Assume that all polygons resulting from the reduction are made thick by one unit along the z axis. Clearly, this shows that Theorem 2.1 holds even for rectilinear polyhedra. We can even extend Theorem 2.3 to prove NP-hardness of computing minimum hulls for convex polyhedra in a similar manner.

These reductions are not very interesting, because they require an arbitrary number of polyhedra in the input. In a future paper we shall show that computing minimum hulls is NP-hard *even for only two polyhedra*, of which one may even be convex. A preliminary version of this result may be found in [8].

4. Approximation algorithm

In the following sections we describe various algorithms for the two-dimensional rectilinear version of the problem. Unfortunately, we have not yet designed efficient

algorithms for the nonrectilinear case. To start with, in this section we describe a polynomial algorithm which constructs valid hulls that closely approximate minimum rectilinear hulls, in a sense described below. In the next section we provide an efficient implementation, and also describe some applications. In Section 6 we describe an exact algorithm for the problem which runs in time exponential in the number of polygons.

The approximation algorithm we describe here is essentially greedy in nature. It iteratively performs local modifications to the shapes of the obstacles, such that each iteration reduces the number of vertices. Let P be a set of pairwise disjoint polygons. For notational convenience, henceforth $MinHull(P)$ will denote a *MinHull* of P. A formal input and output specification of the algorithm is as follows:

Input: P, a set of pairwise disjoint polygons.

Output: A valid hull set H as defined by the *Valid-Hull* predicate, such that,

$$\#(P) - \#(H) \geq \tfrac{1}{2} \cdot [\#(P) - \#(MinHull(P))].$$

We now describe the operations that locally modify the shapes of obstacles. There are two such operations, and both are the major iterative operations of the algorithm.

$P' := Fill\text{-}Well(P)$: Consider any polygon p_i of P. Suppose the boundary of p_i has a shape similar to Fig. 24. By this we mean that there should be two points a and b along the polygon's boundary sharing the same x (or, alternatively, y) coordinate such that either a or b is a vertex of the polygon and the segment $[a, b]$ is outside the polygon. The polygonal region enclosed by p_i's boundary between a and b and the segment $[a, b]$ is called a *well*. *Fill-Well* identifies a well that does not intersect with any other polygon, and *fills* it by replacing the fragment of the polygon's boundary between a and b by the straight line $[a, b]$. Clearly, if a well does get filled, then $\#(Fill\text{-}Well(P)) < \#(P)$.

$P' := Fill\text{-}Corner(P)$: Consider any polygon p_i of P. Suppose the boundary of p_i has a shape similar to Fig. 25. By this we mean that there should be two vertices a and b along the polygon's boundary and a third point d exterior to the polygon such that the segments $[a, d]$ and $[b, d]$ are outside the polygon and perpendicular to each other. The polygonal region enclosed by p_i's boundary between a and b and the segments $[a, d]$ and $[b, d]$ is called a *corner*. Note that a corner is a generalization of the *block*, used in the proof of Theorem 2.1. *Fill-Corner* identifies a corner that does not intersect with any other polygon, and fills it by replacing the fragment of the

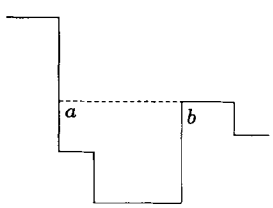


Fig. 24.

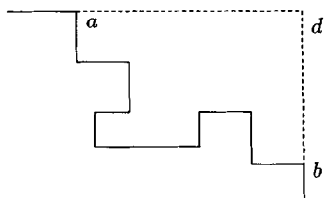


Fig. 25.

polygon's boundary between a and b by the two lines $[a, d]$ and $[b, d]$. If a corner does get filled, then it is easy to see that $\#(\textit{Fill-Corner}(P)) < \#(P)$.

It is easy to see that a *MinHull* can be constructed by applying a particular sequence of these operations. Unfortunately, it is unlikely that the exact sequence can be determined by polynomial time. However, the polynomial-time algorithm we develop applies these operations in a sequence that guarantees the error bound claimed above in the approximate solution. Later in this section, we describe an efficient implementation of this algorithm which runs in $O(n \log n)$ time, where n is the number of vertices in the input.

We are now ready to describe the algorithm, which has two major steps. In the first step wells are filled, and in the second step corners are filled.

Step 1: In this step, *Fill-Well* operations are applied exhaustively on P, converting it to P_1. At any stage, the selection of the well to be filled is arbitrary. Thus, P_1 should have no further wells that can be filled. We claim that *Fill-Well* operations are harmless, that is, $\#(\textit{MinHull}(P)) = \#(\textit{MinHull}(P_1))$. To see that this is true, let the sequence of polygon sets produced during Step 1 be $P = Q^0, Q^1, Q^2, \ldots, Q^b = P_1$. Consider $\textit{MinHull}(Q^i)$ of a set of polygons Q^i. In the worst case it is conceivable that it intersects the next well to be filled as shown in Fig. 26. However, we can construct a new $\textit{MinHull}(Q^{i+1})$ with the same number of vertices which avoids the well, as in Fig. 27.

Step 2: In this step, *Fill-Corner* operations are applied exhaustively on P_1, converting it to P_2. It is easy to see that P_2 should have neither wells nor corners that can be filled. Unlike Step 1, however, at any stage the selection of the corner to be filled is *not*

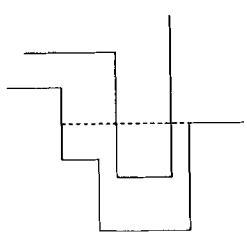


Fig. 26.

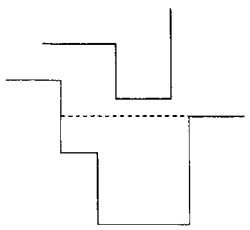


Fig. 27.

arbitrary. The selection procedure is the key idea of the algorithm, and we describe it after the following definitions and facts.

We recall that a staircase is a fragment of a polygon's boundary, composed of a sequence of steps. Unlike convex polygons, a general simple polygon may have more than four staircases. Let us call those fragments of a polygon's boundary that are not staircases, *fixed fragments*, a name we will justify later. Thus, a polygon's boundary consists of alternating staircases and fixed fragments. Consider a polygon with no wells to be filled, such as the one belonging to P_1. Suppose the polygon has a corner that can be filled. Then the common boundary between the corner and the polygon has to be a sequence of adjacent steps of some staircase. This fact has the following important consequence. Consider a set of polygons with no wells to be filled, such as P_1. Then *all* its fixed fragments will be portions of the boundary of any of its tight minimum hulls. Thus, the algorithm has to only examine and modify staircases. Define the *size* of a staircase as the number of steps it contains. Also, a step at either end of a staircase is called a *head*. Clearly, a head is adjacent to some fixed fragment. Figure 28 is an illustration of some of the above notions.

Step 2 of the algorithm starts off as follows. First, a set of fixed fragments is initialized by collecting those fragments of the polygon boundaries that are not staircases. Second, a set of staircases is also initialized by collecting all staircases of the polygons. After this, a corner is selected to be filled, as described below. This will result

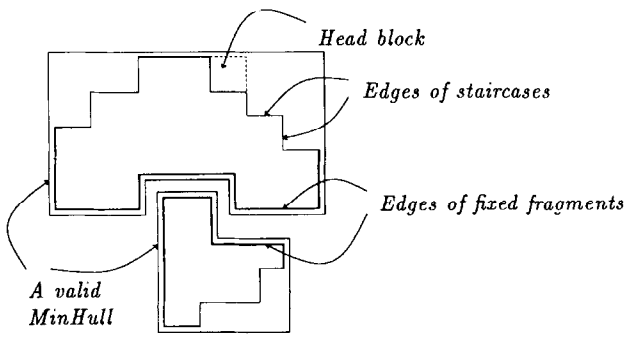


Fig. 28.

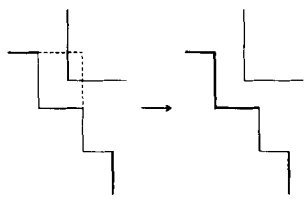


Fig. 29.

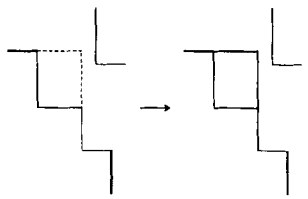


Fig. 30.

in staircases shrinking and fixed fragments growing. The iterations terminate when all staircases have vanished.

The selection procedure in Step 2 can now be described. A head of any staircase is examined to determine if the corner that it defines can be filled. (In the terminology of the proof of Theorem 1, this is equivalent to asking whether the *block* defined by the head step intersects with any other polygon). Two cases arise.

Case 1: The corner cannot be filled. Then the head is clearly a part of the final hull, and is thus removed from the staircase (Fig. 29) and added to the adjacent fixed fragment.

Case 2: The corner can be filled. Then the corner is filled, and the updates made to the staircase and adjacent fixed fragment are indicated in Fig. 30.

We observe the following facts. In either case the size of a staircase is decreased; thus, the iteration of the selection procedure will terminate. Second, one can think of the fixed fragments as portions of partially constructed hulls. At every iteration we add to this partial hulls, until all the staircases vanish. The name *fixed* is now justified because once an edge is added to a fixed fragment, it becomes a part of the final hulls.

The result of the iteration, P_2, may have to be finally tightened to form H, which is the output of the algorithm.

Clearly, the whole process takes polynomial time. The following lemma proves that H indeed satisfies the claimed error bound.

Lemma 4.1. $\#(P) - \#(H) \geq \frac{1}{2} \cdot [\#(P) - \#(MinHull(P))]$.

Proof. From Step 1, we know that

(1) $\#(P) \geq \#(P_1)$,

(2) $\#(MinHull(P)) = \#(MinHull(P_1))$.

Since P_2 does not have any wells and corners to be filled,

(3) $\#(P_2) = \#(MinHull(P_2))$.

At the end of Step 2, we also get

(4) $\#(H) = \#(P_2)$.

Now, let the total number of *Fill-Corner* operations be c. Let the sequence of polygon sets produced during Step 2 be $P_1 = R^0, R^1, \ldots, R^c = P_2$. Consider $MinHull(R^i)$ of a set of polygons R^i. In the worst case it is conceivable that it intersects the next corner to be filled as shown in Fig. 31. However, we can construct a valid new hull for R^{i+1} which has at most 2 more vertices than $MinHull(R^i)$, by "bending" the intersecting portions of $MinHull(R^i)$ outwards to avoid the corner, as shown in Fig. 32. But after filling the corner, the new set of polygons R^{i+1} has at least 2 vertices less than R^i. We, thus, get two more conditions,

(5) $\#(MinHull(P_2)) \leq \#(MinHull(P_1)) + 2c$,

(6) $\#(P_2) \leq \#(P_1) - 2c$.

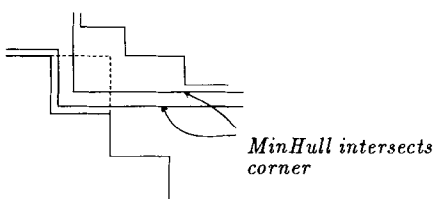


Fig. 31.

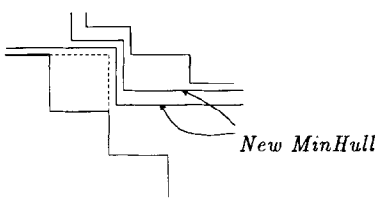


Fig. 32.

In the six conditions above, there are 8 quantities, 3 of which appear in the statement of the lemma. If we eliminate the other 5 quantities, the lemma follows. □

5. Implementation and applications

In this section we provide an efficient implementation of the approximation algorithm outlined in the previous section.

Without loss of generality, we assume that horizontal and vertical edges of the input have distinct y and x coordinates, respectively. We first enclose the set of polygons in a rectangular *room*. A valid boundary for this room can be easily determined in linear time. The *obstacle-free space* is the bounded region within the room which excludes the polygons. The algorithm will try to fill wells and corners, thus fattening the obstacles and shrinking the free space.

The most time-consuming procedure required by the algorithm is to organize the free space into a convenient data structure. The procedure is called a horizontal (or vertical) *rectangulation* of the free space. Horizontal rectangulation involves breaking up the free space into rectangles by extending each horizontal polygon edge within the free space, possibly in both directions, until it hits a vertical edge of some polygon or the room boundary. Clearly, $O(n)$ rectangles will be created by this process. Rectangles are classified as *domestic* if the two vertical sides belong to the same polygon, and *alien* otherwise. The rectangles are then organized as a planar *free-space graph*, where vertices are rectangles, and an edge exists between two vertices if the corresponding rectangles are adjacent. Clearly, the number of edges in this graph is also $O(n)$. Vertical rectangulation is similarly defined. Figure 33 is an illustration of the above notions. Rectangulation is closely related to *trapezoidization* of polygons with holes [9], and can be achieved in $O(n \log n)$ time by a plane sweep. It is easy to implement because it uses elementary data structures such as balanced binary search trees. Except for a few rectangulations, our algorithm runs in linear time.

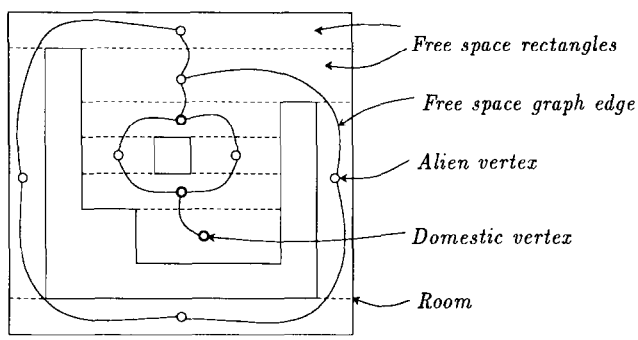


Fig. 33.

Implementation of Step 1: We distinguish between two types of wells that are filled. The wells in Fig. 23 are *horizontal* because they are filled by replacing a portion of the boundary with a horizontal line. *Vertical* wells are similarly defined. We first fill all horizontal wells, then fill all vertical wells. To fill horizontal wells, first perform a horizontal rectangulation and construct the free-space graph. We then traverse the graph as follows, and remove rectangles that correspond to horizontal wells.

The graph traversal performs a procedure called *Visit* at each vertex. Consider any vertex v. The procedure checks whether v is domestic *and* has degree one. If so, then v is removed from the graph, and *Visit* is invoked recursively at the vertex adjacent to v prior to removal. After this, the procedure is invoked at one of the remaining unvisited vertices, and so on. This traversal ensures that rectangles are removed first from the bottom of wells.

Clearly, the total number of visits is linear in the size of the graph, which is $O(n)$. At this stage all horizontal wells have been filled. The shrunken free space is then vertically rectangulated and all vertical wells filled. Thus, Step 1 takes $O(n \log n)$ time, dominated by the two rectangulations.

Implementation of Step 2: The staircases and fixed fragments are maintained as linked lists. We perform a horizontal rectangulation of the free space. The free-space graph will not be necessary here. Instead, each vertical edge of a staircase contains a pointer to the list of rectangles that lie along its side. Recall that in the earlier outline of Step 2, any corner that got filled was defined by the head step of some staircase. The actual implementation is more optimized, and we illustrate by an example. Let the iteration procedure select some staircase, as the one in Fig 34. Instead of just the head step, we run down the staircase from the head and determine the maximum number of adjacent steps defining a corner that can be filled, as in Fig. 35. To do this, in our

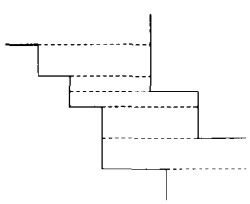


Fig. 34.

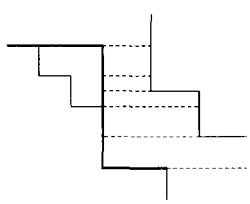


Fig. 35.

example, we keep track of the minimum x coordinate of the right-hand sides of all the rectangles encountered while running down the staircase. We stop just before the x coordinate of the staircase becomes greater. The corner then gets filled, and the updates are straightforward except that some rectangles may become shorter. Since each rectangle is shared by at most two staircases, the time taken for filling all corners excluding the rectangulation is $O(n)$.

Finally, we have to tighten the hulls. We first push the vertical edges of the hulls inwards. To do this, we horizontally rectangulate the region between each original obstacle and its enclosing hull. Clearly, the amount by which any vertical edge of a hull needs to be pushed is the minimum length of all rectangles that lie along its side. A similar procedure is adopted for pushing horizontal edges of hulls. Again, the time is dominated by rectangulations, which is $O(n \log n)$. (Actually, the rectangulations here can be accomplished in $O(n \log \log n)$ time using the algorithm given in [19], or even in $O(n)$ time using a recent algorithm given in [5], because the regions to be rectangulated are polygons with single holes.)

Thus, the whole algorithm runs in $O(n \log n)$ time.

We describe an application of the above algorithm. Consider the problem of computing the shortest rectilinear path between two given points s and t which avoids a set of rectilinear polygonal obstacles. This problem appears in circuit design (wire routing) as well as robot motion planning. We will show that our algorithm may speed up the process of computing such paths. First, we will establish that the lengths of rectilinear shortest paths are preserved, even if we replace obstacles with valid hulls.

Lemma 5.1. *Let E be a plane which contains a set of disjoint rectilinear polygons, P. Let H be a valid set of hulls of P. Let s and t be two points within $E-H$. Then the shortest rectilinear distance between them within $E-H$ is equal to the shortest rectilinear distance between them within $E-P$.*

Proof. For any i, consider the shapes of the fragments of the region $h_i - p_i$. Because all edges of h_i touch the boundary of p_i, these fragments will assume shapes similar to the ones shown in Figs. 36 and 37. Now consider a shortest path between s and t within $E-P$. It, clearly, cannot intersect a fragment of the Fig. 36 type, unless it lies along the boundary of h_i. It can, however, intersect a fragment of the Fig. 37 type. But as the figure shows, the intersecting portion can be replaced by nonintersecting portion of

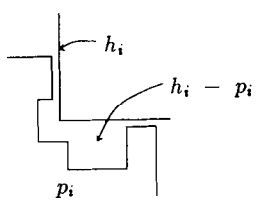


Fig. 36.

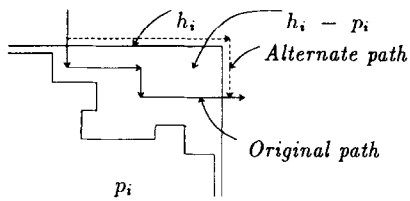


Fig. 37.

the same length. Thus, we can replace all intersecting portions of the shortest path with nonintersecting portions such that the new path lies wholly within $E-H$, and its length is the same as that of the original shortest path. □

Now consider the problem of computing the shortest rectilinear path between s and t, amidst the rectilinear obstacles P. We first regard s and t as additional point obstacles, and use our approximation algorithm to construct approximate minimum hulls of the obstacles, in time $O(n \log n)$. After this preprocessing, any shortest-path algorithm can be employed, such as the one in [6], which runs in $O(N(\log N)^2)$ time, where N is the *reduced* number of vertices.

6. Exact algorithm

Given a rectilinear polygon set with k polygons and a total of n vertices, we describe an algorithm that computes their *actual* minimum hulls with $O(n^{2k+9})$ running time. The algorithm works even for nonconvex rectlinear polygons. This result shows that the difficulty of the problem is related more to the *number of polygons* than to their shapes and spatial positions.

Without loss of generality, let us assume that in the input, no two horizontal (vertical) edges share the same y coordinate (x coordinate). Suppose we *grid* the area exterior to the polygons by extending vertical and horizontal lines through each vertex in both directions until they either extend to infinity, or terminate at some polygon's boundary. It is not hard to see that there exists a set of minimum hulls whose edges lie along the grid. One obvious algorithm to compute minimum hulls is to perform a brute force search on this grid. However, the time will be exponential in n. To achieve the claimed time bound, we need to do better.

The algorithm is as follows. Let $P=\{p_1, \ldots, p_k\}$ be the set of polygons. First fill all vertical and horizontal wells. Then confine the resulting polygons in a rectangular room. Next, compute the midpoint of the leftmost vertical edge of every polygon p_i. From every such point, draw a horizontal line segment called l_i through the free space to the left, until it terminates at a vertical edge of some polygon or the room. These segments are called *partition edges*.

If we superimpose our grid on these edges, we observe that each partition edge is intersected by possibly $O(n)$ vertical grid lines. If we are constructing tight hulls, then *only two* of the $O(n)$ grid points on each partition edge are actually intersected by the hulls. Let us fix an arbitrary pair of these grid points for each partition edge, thus fixing $2k$ points altogether. Suppose the problem now is to compute minimum hulls *subject* to the constraint that they are required to intersect these $2k$ points. As we shall shortly show, this subproblem can be solved in $O(n^9)$ time. Now our original problem is easily solved. For each combination of $2k$ grid points, we solve the constrained problem and, finally, output the hulls with minimum vertices. Since every combination has k pairs of points, with each pair being selected from at most $O(n)$ points, there are at most $O((n^2)^k)$ combinations. Thus, the running time of our algorithm is $O(n^{2k} \cdot n^9) = O(n^{2k+9})$. Clearly, the most time-consuming process in our algorithm involves going through all the combinations of $2k$ points each.

We now show that the constrained subproblem can be done in polynomial time, which is the main idea of this section. Our algorithm for this is based on dynamic porgramming. We first build up a discrete structure for the algorithm to search.

Consider the free space with the partition edges drawn. Let us assume that each partition edge is actually drawn as a pair of closely spaced parallel lines. Then they may be imagined as narrow *channels* connecting the interiors of pairs of polygons (or the interiors of polygons with the exterior of the room). Clearly, the free space will then resemble a *single* polygon (Fig. 38). Horizontally rectangulate this polygon, and build up a *free-space graph* exactly as in the previous section. Thus, rectangles are vertices, and edges are between adjacent rectangles. Since the free space is a simple polygon, the graph is clearly a *tree*. Let us root the tree at some vertex. We observe that the leaves are those rectangles which have one of the partition edges as a side.

Note that the horizontal edges of the rectangles may be composed of portions that are polygon edges, and portions that are extensions of polygon edges in the free space. The latter portions have been constructed by the rectangulation process, and are known as *free edges*. As Fig. 39 shows, a rectangle may have two, three, or even four

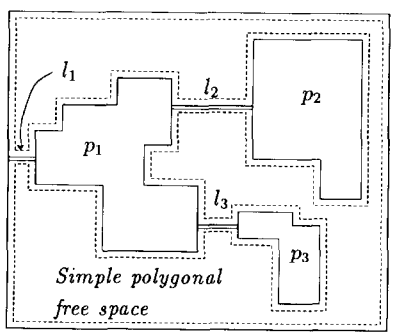


Fig. 38.

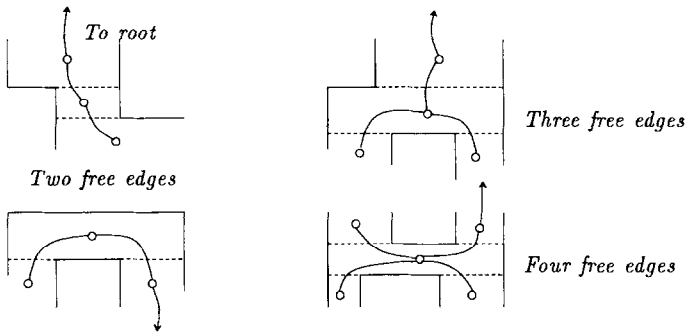


Fig. 39.

free edges. For rectangle r (other than the root), let e_r denote the free edge between r and its parent rectangle in the free-space free.

If we superimpose the grid on the rectangulation, we observe the following. There is a set of minimum hulls whose horizontal edges run along the horizontal edges of the rectangulation. However, we will follow the convention that these edges do not run directly along the rectangle edges, rather they run in parallel, with a slight gap. This ensures that the vertices of the hulls are formed within the rectangles, rather than on their boundaries. As with the partition edges, each free edge is intersected by possibly $O(n)$ vertical grid lines. Furthermore, of the $O(n)$ grid points, only two are actually intersected by the hulls. Our convention also ensures that the vertical hull edges do not terminate at these points, rather they pass through them.

We are now ready to describe the polynomial-time algorithm for solving the constrained-minimum-hulls subproblem. Assume that a pair of grid points has been fixed for each partition edge. The overall idea is to explore the tree bottom up, and at any stage, compute all possible minimum-hull fragments within the region visited thus far. The algorithm is based on dynamic programming, because previous computations further down the tree are stored and later used in computations up the tree. When the root is reached, the final minimum hulls are computed.

In more detail, the algorithm visits a rectangle only if all its children have been visited. The purpose of visiting rectangle r (other than the root) is to compute an $O(n) \times O(n)$ matrix M_r, which is described as follows. Let R_r refer to the polygonal region formed by the union of r and all its children. The entry $M_r[i, j]$ (where $i<j$) contains the fragments of the hulls with *minimum vertices* within the region R_r such that

(1) these hulls intersect e_r at its ith and jth points, and

(2) for each partition edge at the leaves of R_r, these hulls intersect at the corresponding pair of fixed grid points.

We shall now show how to compute this matrix while visiting a vertex r. Clearly, r has one, two, or at most three children, and their respective matrices have already been computed. We describe only the case where r has three children. The other cases are even simpler; therefore, we do not describe them here.

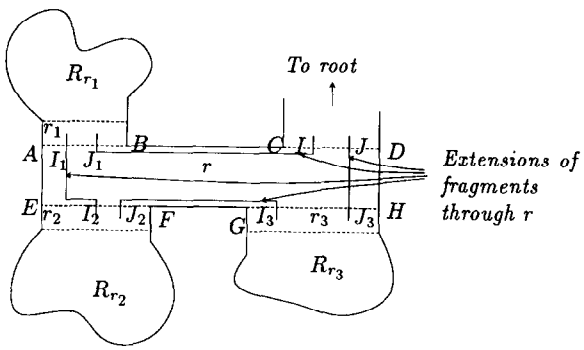


Fig. 40.

Consider Fig. 40, where the children of r are r_1, r_2 and r_3, and $e_r = CD$, $e_{r_1} = AB$, and so on. Suppose we are trying to compute the entry $M_r[i,j]$. Let the ith and jth grid points of e_r be I and J, respectively. Select three pairs of integers, (i_1,j_1), (i_2,j_2), and (i_3,j_3). Let the i_1th and j_1th grid points of e_{r_1} be I_1 and J_1, respectively. Similarly, let the i_2th and j_2th grid points of e_{r_2} be I_2 and J_2, respectively. Finally, let the i_3th and j_3th grid points of e_{r_3} be I_3 and J_3, respectively.

Suppose we are to extend the hull fragments in $M_{r_1}[i_1,j_1]$, $M_{r_2}[i_2,j_2]$, and $M_{r_3}[i_3,j_3]$ through the rectangle r such that they intersect e_r at I and J. The way to do this involving the least number of additional vertices is as follows. Because we are dealing with tight hulls, it is clear that J_1 (J_2) has to be connected to I (I_3). This involves four additional vertices. Similarly, I_1 has to be connected to I_2. If they do not have the same x coordinate, this involves two additional vertices. Finally, J_3 has to be connected to J. If they (fortunately) have the same x coordinate, this can be done without any additional vertices. Thus, in this situation, six additional vertices are required.

Now the method of computing $M_r[i,j]$ is clear. For *every* three pairs of integers, (i_1,j_1), (i_2,j_2), and (i_3,j_3), consider the hull fragments present in $M_{r_1}[i_1,j_1]$, $M_{r_2}[i_2,j_2]$, and $M_{r_3}[i_3,j_3]$. Extend them through r with the least number of additional vertices, until they intersect e_r at I and J. Once this is done for all possible three pairs of integers, select the fragments (with their extensions) which have the least total number of vertices and store them in $M_r[i,j]$. Thus, all entries of the matrix can be filled in this way.

This process of computing all possible minimum-hull fragments can be continued up the tree until we come to the root rectangle. The root has at most four children. Let us describe the case when the root has exactly four children, the other cases being even simpler. The processing here is very similar to the extensions of hulls described earlier. We select four pairs of integers, select the corresponding hull fragments from the children matrices, and appropriately patch them within the root rectangle, using the least number of additional vertices. This is carried out for every selection of four pairs

of integers, and the hulls with the minimum vertices over all others are the final constrained minimum hulls.

We show that this constrained-hulls algorithm runs in polynomial time. There are $O(n)$ rectangles in the tree. For rectangles that are not the root, hull fragments are extended at most $O(n^8)$ times, and each extension can be accomplished in constant time. For the root, hull fragments are also patched at most $O(n^8)$ times, and each patchwork takes constant time. So the total time taken is $O(n^9)$.

7. Future directions

A number of interesting open problems are raised by this work. In the two-dimensional case, is the problem NP-hard even if only a bounded number of nonrectilinear polygons is allowed in the input? We think that this problem is solvable in polynomial time. In fact, we have shown in this paper that the rectilinear problem with a bounded number of polygons is solvable in polynomial time. However, we have recently shown in [8] that the three-dimensional problem is NP-hard even for a bounded number of polyhedra.

In our approximation algorithm, can we get better that a factor of half in the number of vertices reduced? Perhaps, a more judicious selection of *Fill-Corner* operations will help.

How efficient is our approximation algorithm? If any algorithm is based on rectangulations, it will require $\Omega(n \log n)$ time, because that is the lower bound for computing rectangulations [9]. However, there may exist completely different efficient methods that avoid rectangulations, and yet achieve a factor of half in the number of vertices reduced.

We do not have approximation algorithms for nonrectilinear polygons. Our rectilinear algorithm cannot be generalized, because it is dependent upon the fact that at most two corners of polygons can simultaneously intersect. In the general case, however, any number of blocks can simultaneously intersect.

How good is our exact algorithm for computing minimum rectilinear hulls? Even though its time complexity is exponential in k, the factor of n^9 makes it impractical even for a few polygons. We believe that the algorithm for constrained minimum hulls can be dramatically improved to run in some low-degree polynomial time.

We do not have an exact algorithm for nonrectilinear polygons which is exponential only in the number of polygons. The major problem here seems to be in constructing a suitable discrete search space.

In three dimensions, we need to design algorithms for all variants of the problem. Unfortunately, most of our ideas in two dimensions do not translate there, and radically different approaches may be required.

Finally, it seems to be necessary to define a good *general measure of approximation*, rather than special ones such as area, number of vertices or faces, etc. Such a measure should be investigated from a computational complexity point of view.

References

[1] A. Aggarwal, H. Booth, J. O'Rourke, S. Suri and C.K. Yap, Finding minimal convex nested polygons, in: *Proc. ACM Symp. on Computational Geometry* (1985) 296–303.

[2] H. Alt, Approximation of convex figures by circles and rectangles, Manuscript, 1989, Freie Universität Berlin.

[3] J. Canny and J. Reif, New lower bound techniques for robot motion planning, in: *Proc. IEEE FOCS* (1987) 49–60.

[4] J.S. Chang and C.K. Yap, A polynomial solution for Potato Peeling and other polygon inclusion and enclosure problems, in: *Proc. IEEE FOCS* (1984) 408–417.

[5] B. Chazelle, Triangulating a simple polygon in linear time, in: *Proc. IEEE FOCS* (1990) 220–230.

[6] K. Clarkson, S. Kapoor and P.M. Vaidya, Rectilinear shortest paths through polygonal obstacles in $O(n(\log n)^2)$ time, in: *Proc. ACM Symp. on Computational Geometry* (1987) 251–257.

[7] G. Das and D. Joseph, Minimum vertex hulls for polyhedral domains, in: *Proc. STACS, 1990*, Lecture Notes in Computer Science, Vol. 415 (Springer, Berlin, 1991) 126–137.

[8] G. Das and D. Joseph, The complexity of minimum convex nested polyhedra, in: *Proc. Canadian Conf. on Computational Geometry* (1990) 296–301.

[9] A. Fournier and D.Y. Montuno, Triangulating simple polygons and equivalent problems, *ACM Trans. Graphics* **3** (1984) 135–152.

[10] H. Fraysseix, J. Pach and R. Pollack, Small sets supporting fary embeddings of planar graphs, in: *Proc. ACM STOC* (1988) 426–433.

[11] M. Garey and D. Johnson, *Computers and Intractability* (Freeman, New York, 1979).

[12] S.K. Ghosh and A. Maheshwari, An optimal algorithm for computing a minimum nested nonconvex polygon, Tata Institute of Fundamental Research Report TR CS-90/2, TIFR, Bombay, 1990.

[13] V. Klee, private communication with D. Joseph, 1990.

[14] V. Klee and M.C. Laskowski, Finding the smallest triangle containing a given convex polygon, *J. Algorithms* **16** (1985) 359–374.

[15] D. Lichtenstein, Planar formulae and their uses, *SIAM J. Comput.* **11** (1982) 329–343.

[16] J. O'Rourke, Computational geometry column, *SIGACT News* **19** (1988) 22–24.

[17] M. Sharir and P. Schorr, On shortest paths in polyhedral space, in: *Proc. ACM STOC* (1984) 144–153.

[18] S. Suri, A polygon partitioning technique for link distance problems, Manuscript, Bell Communications Research, 1986.

[19] R.E. Tarjan and C.J. Van Wyk. An $O(n \log\log n)$-time algorithm for triangulating a simple polygon, *SIAM J. Comput.* **17** (1988) 143–178.

[20] C.A. Wang, Finding a minimal chain to separate two polygons, in: *Proc. Allerton Conf. on Communication, Control and Computing* (1989) 574–583.