Optimal Polygon Placement on a Grid

By

Dominic Lessard

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute for Computer Science
School of Computer Science
Carleton University
Ottawa, Ontario

April 30, 2000

Abstract

In this thesis we study the following geometric optimization problem. Given a simple polygon P and a grid, we want to find a placement of P on the grid such that there are a maximum number of intact squares inside P. We present several algorithms and data structures to solve this problem efficiently. There are different variants of the problem depending on the operations that can be applied to P. We solve the problem when only x-translations of the polygon are allowed, and also for general translations. We study the case where the grid is a set of unit squares, but our algorithms can be used for other grids, such as triangular and hexagonal grids. Worst case analysis of our algorithms are presented and optimality of solutions found are proven. This research is joint work with Dr. Bose and Dr. Czyzowicz.

Acknowledgments

Je tiens à remercier mes superviseurs, Dr. Bose et Dr. Czyzowicz pour leur disponibilité, leur générosité et leur collaboration. Ils ont été une source d'inspiration sans laquelle ce mémoire n'aurait pu être complété.

Je remercie aussi mes parents pour leur soutien moral et financier tout au long de mes études. Ce ne fut certainement pas évident de voir partir votre plus vieux, loin de la campagne. Merci.

I also want to thank all the people in PARADIGM Group for their help and useful discussions. Many thanks to other graduate students and to the staff at the main office of the School of Computer Science. You are the best! I spent two great years at Carleton that won't be forgotten.

Merci aux amis, colocs et collègues, plus particulièrement à Marie-Claude, Miguel, Jean-Denis, Véronique, Julie, Isabelle, Alain, Viviane, Steven, André, Maryse et Lucette. Un dernier remerciement à ceux qui ont contribué directement ou non, volontairement ou non, à ma décision de partir pour Ottawa. Ce fut la bonne.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

The use of geometry based programs has increased recently, motivated by geographic information systems, computer graphics, CAD/CAM, VLSI circuit design, optimization problems, data compression, manufacturing processing, pattern recognition and image processing.

In the field of computational geometry, researchers solve geometric problems by designing efficient algorithms. Worst case analysis, expected case analysis and experimental simulations of their algorithms and data structures are used in order to compare different methods. In this thesis our research focuses on the following geometric optimization problem. The general version of the problem:

Problem 1.1 *Given a simple polygon P and a set of non-overlapping squares S, place P on S such that the number of whole squares in P is maximized.*

The problem has many variants. The squares may have different sizes and orientations. Operations such as translation, rotation and scaling may be applied to the set of squares, as well as to the polygon. Also, one might change the optimization criteria (e.g. maximizing the area of the squares inside polygon). Examples of solutions to different variants of Problem 1.1 are shown in Figure 1.1.

A placement of a simple polygon on a grid is shown in (A). If the set of squares is given, then placements look like the ones shown in (B) or (D). Another problem

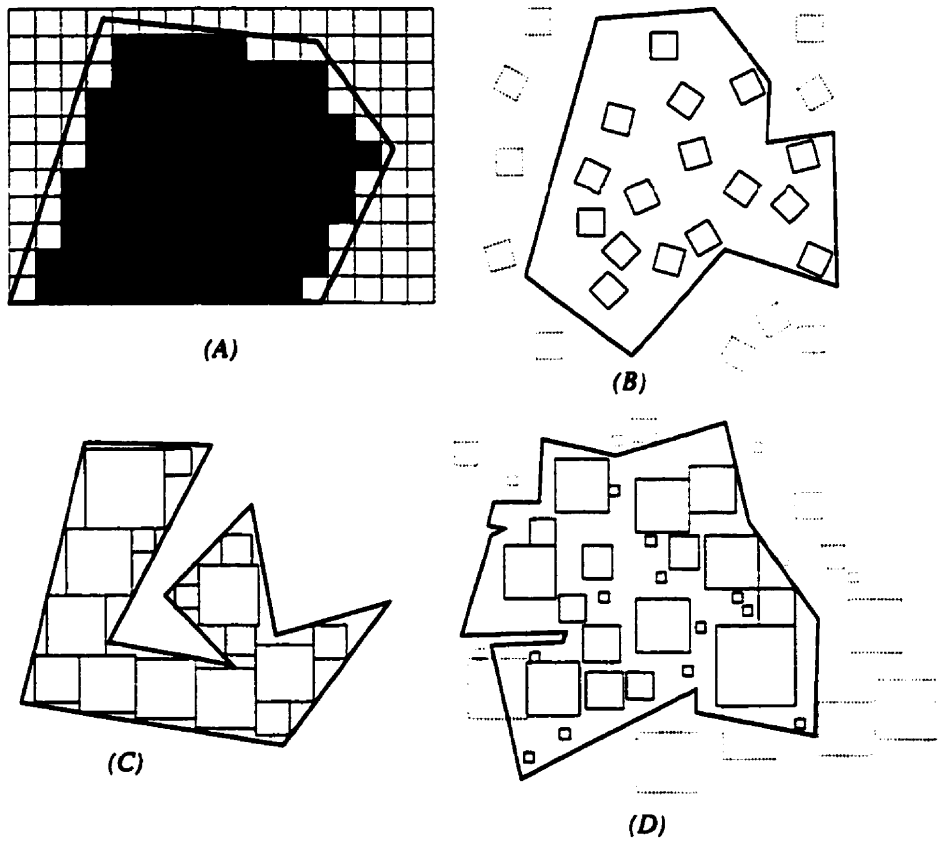

Figure 1.1: Examples of solutions to problem 1.1.

is to maximize the area covered by a set of squares inside a simple polygon (C). The squares have size at least k, and they do not overlap.

In this thesis, we concentrate on the set of squares coming from a grid, as shown in Figure 1.1 (A). In essence, we want to tile a polygon such that the number of intact squares in the polygon is maximum. We will solve these problems efficiently using geometric techniques. A related problem, is to minimize the number of tiles needed to cover completely a polygon.

1.1 Motivations

First, the containment problem that we study in this thesis, to our knowledge, has not been addressed previously. The main application is covering floors of complex shaped rooms. Depending on the optimization criteria and the constraints, our methods can be applied in some parts of the computation. In fact, one might want to minimize the total number of bricks to cover a pavement. Our solution permits to maximize the number of intact squares inside a polygon, which minimizes the area to cover using partial squares.

1.2 Summary of Results

Given a simple polygon P with m edges, height h and width w, we want to find a placement of P over a grid of unit squares maximizing the number of grid squares in P. First, we study the problem where only x-translations of P are allowed. In this case, we show that an optimal solution can be found in $O(m + h \log m)$ if P is convex, and $O((m + hl) \log m)$ if P is nonconvex where l is the number of locally minimal vertices of P. However, if $l = 0$, then our algorithm runs in $O((m + h) \log m)$ time. Second, we study the problem where x and y translations of P are allowed. We show that an optimal solution can be found in $O((m + h^2 + w^2) \log(m + h + w))$ time when P is convex, and $O(m(h + w) + (m + r^2(h^2 + w^2)) \log(m + rh + rw))$ time otherwise, where r is the number of reflex vertices.

1.3 Outline of Thesis

The outline of the thesis is as follows. The next chapter discusses related work and industrial applications. Some definitions, preliminaries and a few basic algorithms and data structures appear in Chapter 3. In Chapter 4 we show how to sort grid points by their horizontal distances from a line segment in linear time, a key step in the efficiency of our algorithms. Chapter 5 is the core of the thesis. The reader will find the pseudo-code of our placement algorithms as well as their complexity analysis.

Each section of that chapter focuses on a particular case of the problem. Algorithms that we present can be applied to different type of grids. Examples are shown in Figure 1.2. Finally, the last chapter of the thesis discusses some open problems.

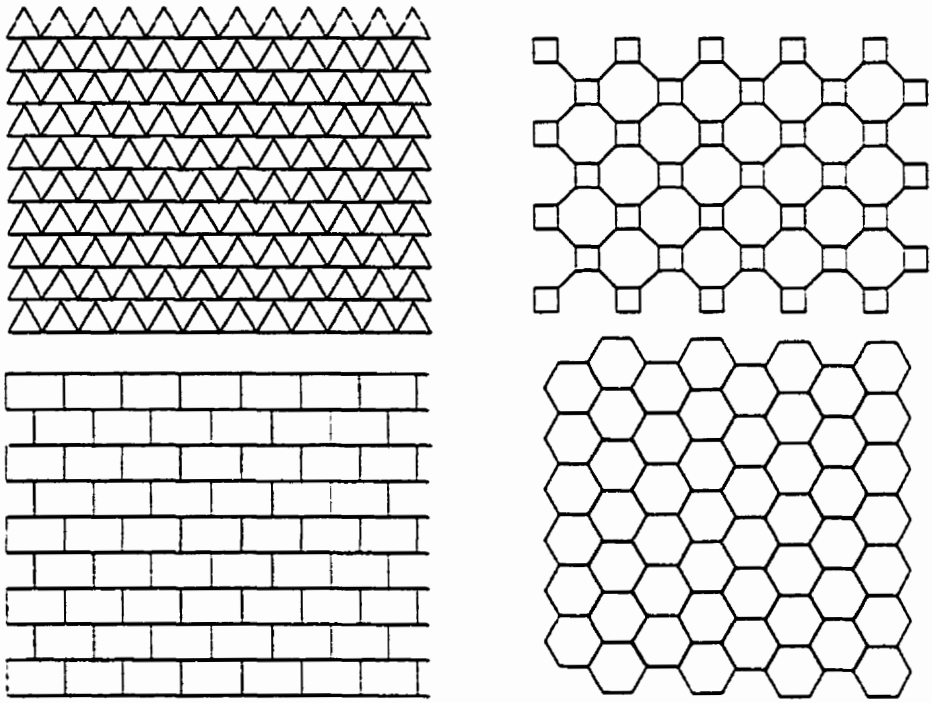

Figure 1.2: Different grids.

Chapter 2

Review of the Literature

The main application motivating this work is covering floors with a total minimum number of tiles. Depending on the context, floors can be replaced by other surfaces such as parking lots or paved stones. In this chapter we review the results appearing in the literature on tilings, containment and geometric optimization problems. Basic concepts and definitions can be found in many textbooks on algorithms [CLR90, Man89, BB87], computational geometry [BKOS97, BY98, GO97, Mul94, O'R98, PS85] and other textbooks on combinatorial geometry and optimization [PA95, Pac93, Aga91, GLS88].

2.1 Tilings

Grünbaum and Shephard [GS89] has defined and classified tilings and patterns.

1. The formal definition of tiling [GS89] is a countable family of closed sets $\mathcal{F} = \{T_1, T_2, ...\}$ which covers the Euclidean plane without any gaps or overlaps. T_i's are the tiles. The difference between packing and covering is that the first one is a family of sets without any overlap while the second one covers the plane with no gap at all.

2. A monohedral tiling consists of only a set of polygons having the same size and shape.

3. A tiling is edge-to-edge if the polygons which form the tiles coincide with the corners and the sides of the tiling.

4. The only edge-to-edge monohedral tilings by regular polygons are tilings of squares, equilateral triangles or regular hexagons.

Kenyon [Ken97] presented some theoretical results on tilings of convex polygons and on minimizing the number of squares needed to tile a rectangle [Ken96]. Sirbu [Sir92] has shown that plane tilings and their properties have applications in medicine, where tilings are used to describe the fight between the immune system and a pathogen agent. Richard et al. [RHHB98] discussed concepts about random tilings. Paredes et al. [PAB98] stated that tilings with squares and triangles are very useful tools to study several structural and thermodynamical properties of a wide variety of solids. Keating and King [KK99] worked on tilings with squares. They have shown a necessary and sufficient condition for a bounded region of the plane with rectangles to be tillable with finitely many squares. The rectangles have the form $[a_1, a_2) \times [b_1, b_2)$ where $a_1 < a_2$ and $b_1 < b_2$.

2.2 Covering Problems

We say that a problem L is NP-Complete if and only if $L \in NP$, for every $L' \in NP$, and $L' \leq_p L$ (polynomial reduction to L) [CLR90]. It is not known if there exists a polynomial time algorithm solving an NP-Complete problem. If the second condition is met by a problem L but not necessarily the first one, then L is NP-Hard [CLR90].

Culberson and Reckhow [CR94] have proven that covering the interior of a polygon with a minimum number of convex polygons is NP-Hard. They have also shown that the problem of covering the interior or the boundary of an orthogonal polygon with rectangles is NP-Complete. They used reductions from SAT [CLR90, Man89, Mor98] for the first problem and 3SAT for the last one.

Motivated by VLSI mask generation, remote sensing and image compression, Bar-Yehuda and Ben-Hanoch [BB96] presented a linear-time algorithm for covering a simple orthogonal polygon (without holes) with a minimum number of squares.

Levcopoulos and Gudmundsson [LG97] designed an algorithm to cover an n-vertex polygon P with a minimum number of squares in $O(n^2 + \mu(P))$ time, where $\mu(P)$ is the minimum number of squares needed to cover P. The squares may overlap. They also designed an algorithm running in $O(n \log n + \mu(P))$ time where the total number of squares is no more than $11n + \mu(P)$.

Keil [Kei97] published an $O(n \log n + nm)$ time algorithm to cover an orthogonal polygon P with a minimum number of rectangles, where n is the number of vertices in P and m is the number of edges in the visibility graph of P. His objective was to minimize the number of rectangles covering P. Rectangles are allowed to overlap. If two rectangles A and B are overlapping then either $A - B$ or $B - A$ is connected.

2.3 Geometric Optimization Problems

Agarwal and Sharir [AS96] published some randomized algorithms solving three optimization problems, including computing the biggest stick in a simple polygon. In [AS95], they presented a survey on different algorithmic techniques for geometric optimization problems. Hochbaum [Hoc97] presented surveys on approximation algorithms for NP-Hard problems. Mitchell [Mit97] presented some approximation algorithms for geometric network optimization problems.

Fischer and Höffgen [FH94] tackled the problem of computing a maximum axis-aligned rectangle in a convex polygon. Their motivations were applications in pattern recognition [Cha83], since geometric objects are treated as physical entities. Data compression is another possible application. They solved the problem in $O(\log^2 n)$ time, where n is the number of vertices in the polygon. Later, Alt et al. [AHS97] solved the problem in $O(\log n)$ time.

2.4 Points Containment Problems

Problem 2.1 *Given a polygon P and a set S of n points in the plane, find a placement of P such that the number of points of S inside P is maximized.*

Barequet et al. [BDP95] studied this problem when P is a simple polygon and the rotation operation of P is not allowed. Their algorithm runs in $O(nk\log(pk)+p)$ time where n is the number of points, p the number of vertices of the polygon and k is the maximum number of points contained by the polygon.

Applications of this problem include optimal object placement (CAD), clustering and statistical data analysis. Their techniques do not solve our problem directly. In fact, one might think of finding the optimal solution to Problem 1.1 by finding the maximum number of points (vertices of the tiles) inside the polygon. But Figure 2.1 shows how these problems differ. Triangle P_1 contains seven grid points but no square. Below, P_2 contains fewer grid points (four) and one square, which is optimal for our problem. There is no obvious relation between these two solutions since the points lie on the same line for the first solution.

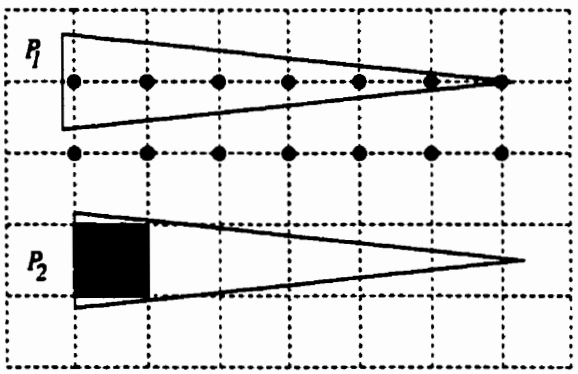

Figure 2.1: Two optimal solutions for two similar problems.

Later, Dickerson and Scharstein [DS98] solved Problem 2.1 where P is convex and both translations and rotations are allowed. Their algorithm runs in $O(n^2kp^2\log(np))$ time and needs $O(n+p)$ space where k is the maximum number of points that P can contain, n is the number of points in the set and p is the number of vertices in P. Thus, their algorithm is output-sensitive.

Akiyama et al. [AIUU96] published a paper where they showed some theoretical results on circles containing maximum number of points. Fischer [Fis93] worked on the problem of finding a maximum area convex polygon whose vertices come from a

set of points marked positive, but not containing any negative marked points, even on its boundary. This problem can be solved in $O(n^4 \log n)$ time where n is the total number of points (positive and negative). Applications are statistical clustering, pattern recognition and data compression.

2.5 Polygon Containment Problems

The following problem has been studied extensively since the early 80's.

Problem 2.2 *Given two polygons P and Q, determine whether or not P can be placed entirely inside Q.*

Chazelle [Cha83] considered both translation and rotation as operations that can be applied to P. Let p and q be the number of vertices of P and Q respectively and $n = p + q$. If P and Q are convex, he constructed an algorithm running in $O(n^3)$. But the complexity is much higher when polygons can be non convex as stated in his paper ($O(n^7)$ time). Chazelle also gave many definitions on contacts and stable placements that are now used in many works on containment problems.

Baker et al. [BFM86] described an algorithm solving the previous problem in $O((q^2+pq)\log(pq))$ time, for P convex and Q non convex respectively. Fortune [For85] presented an algorithm running in $O(pq\log(pq))$ where P is convex and rotation is not allowed.

Later, Avnaim and Boissonnat [AB89] described an algorithm that reports all possible placements of P (subject to translations and rotations) without intersecting another polygon Q. Their method runs in $O(p^3q^3\log(pq))$. They were motivated by cutting stock, inspection and motion planning. Chiu and Wang [CW90] also studied the problem with translations only. They designed algorithms for different variants of Problem 2.2. Grinde and Cavalier [GC96] found an algorithm running in $O(bpq)$ for convex polygons using mathematical programming, b is the number of optimal dual bases encountered while P is rotating from 0 to 2π radians.

Agarwal et al. [AAS98] studied the problem of finding the largest copy of a polygon P inside another polygon Q. Their algorithms runs in $O(pq^2\log q)$ time.

They also have shown that the combinatorial complexity of the space of all similar copies of P inside Q is $O(pq^2)$ and can be computed in $O(pq^2 \log q)$ time.

The following problem has been studied by Grinde and Cavalier [GC97].

Problem 2.3 *Given three simple polygons P_1, P_2 and Q, determine whether or not Q can contain entirely P_1 and P_2 without overlap. Report such a placement if one exists.*

Grinde and Cavalier designed an algorithm which runs in $O(b(p_1 + p_2)q)$ time, where p_1, p_2 and q are the number of vertices of the polygons respectively and b is the number of dual bases encountered. They solved the problem using parametric programming with a nonlinear parameter. Translations and rotations of the polygons were allowed.

Avnaim and Boissonnat [AB87] have worked with one, two and three polygons. They provided general solutions for one and two polygons, but their results regarding three polygons are restricted by the shape of the container, which must be a parallelogram. They also generalized some of their results for higher dimensions.

Ghosh [Gho90] studied some problems related to the Minkowski sum including a solution for the containment problem with translations only. In 1994, Sharir and Toledo [ST94] presented some algorithms solving extremal polygon containment problems. Those include computing a placement for the largest copy of P that does not intersect a set of polygonal obstacles (allowing translation, rotation and scaling), placement of the largest homothetic copies of one or two convex polygons inside another convex polygon and also the problem of finding the placement of the largest similar copy of a triangle in a convex polygon.

Daniels et al. [MDL91, MDL92, Dan95, DM96, DM95, MD95] worked on multiple translational containment, motivated by marker making for clothing manufacture. This application is defined by the following: given a large piece of cloth, maximize the number of clothing parts that can be cut out of the initial cloth. The main goal is to minimize the area of the remaining pieces once the clothes have been cut out from the cloth. They developed many algorithms (geometric and also based on linear programming and mathematical programming) to solve some variants of

the original problem. Later, Milenkovic [Mil98a, Mil98b] worked on the following: given an overlapping layout of k polygons in a container Q, translate and rotate the polygons in order to diminish their overlap to a local minimum.

Martin and Stephenson [MS88] designed algorithms to determine whether or not an object can be put inside another, in two and three dimensions. They also worked with curved polygons. Both translations and rotations are considered in their work.

De Berg et al. [BDK+96] have shown that the maximum number of combinatorially distinct placements of P with respect to Q, both convex, and under translation only is $O\left(p^2 + q^2 + \min(pq^2, p^2q)\right)$. They also designed an $O\left((p+q)\log(p+q)\right)$ time algorithm to find the translation which maximizes the area of the overlap of P and Q.

Chapter 3

Definitions and Preliminaries

Because the optimal placement problem has many variants we define more precisely the problems solved in this thesis. Definitions and basic concepts are also presented. Properties of an optimal placement are discussed in Section 3.4.

3.1 Some Definitions

A polygon is a collection of n vertices $v_1, v_2, ..., v_n$ and n edges $v_1v_2, v_2v_3, ..., v_nv_1$, such that no pair of non consecutive edges share a point. A line segment ab is a closed subset of a line contained between two points a and b. These points are also called its endpoints [O'R98]. Each of these have x and y coordinates. A polygon is convex if its internal angles are at most 180 degrees. A polygon is orthogonal if its edges are parallel to one of the axes of the plane. A vertex at which the internal angle is greater than 180 degrees is a reflex vertex.

Definition 3.1 (Grid) *A grid $\mathcal{G}$ is a set of axis-parallel unit squares. These squares intersect only at their boundaries and do not overlap.*

We use $\mathcal{G}$ to exclusively denote a grid. Although we use unit squares, it is possible to apply a scaling function over the polygon if the square has different size. See Figure 3.2 for the following definitions.

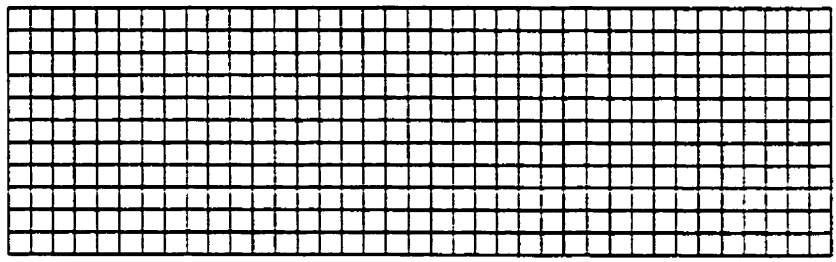

Figure 3.1: Grid.

Definition 3.2 (Supporting wall) *An edge e of a polygon is called a supporting wall if its endpoints lie on the same horizontal line of the grid.*

Definition 3.3 (Supporting vertex) *A grid point is a supporting vertex if it lies on an edge of a polygon and is also a corner of a grid square lying entirely inside it.*

Definition 3.4 (Contact) *A contact between two polygons P and Q occurs if a vertex of P lies on an edge of Q or a vertex of Q lies on an edge of P.*

Problem 3.1 (General) *Given a simple polygon P and a grid $\mathcal{G}$ find a placement of P on $\mathcal{G}$ such that the number of squares inside P is maximized.*

The computation of an optimal solution to this problem allows translation and rotation operations of the polygon. Observe that an optimal solution of the general version of the problem may not have a supporting wall, as shown in Figure 3.3.

Problem 1 (1 degree of freedom) *Given a simple polygon P and a grid $\mathcal{G}$, find a placement of P over $\mathcal{G}$ such that the number of intact squares inside P is maximized, when only x-translation operations are allowed and P must have one of its edges lying on a horizontal line of $\mathcal{G}$.*

Definition 3.5 (Local minimum) *A vertex v of a simple polygon P is a local minimum if its incident edges lie above a horizontal line passing through v.*

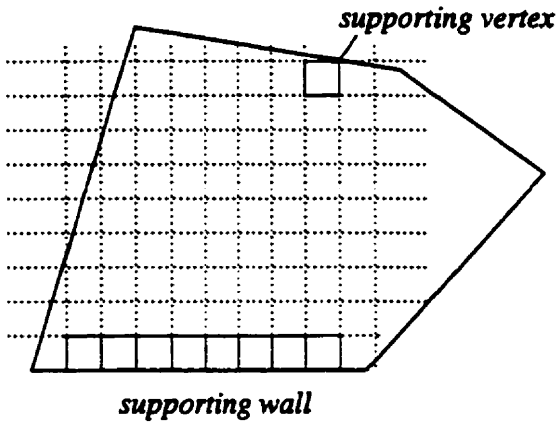

Figure 3.2: Convex polygon with a horizontal supporting wall and a supporting vertex.

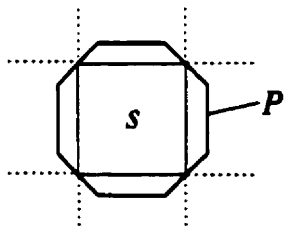

Figure 3.3: Example of an optimal solution without supporting wall.

Figure 3.4 shows a simple polygon with 6 local minima. The number of local minima, denoted l, inside a polygon and its height will be used in the complexity analysis of our algorithms. The running time of our algorithms depend on the number of local minima and the number of intersection points with horizontal lines of the grid.

The following theorem shows how to compute the area of a simple polygon provided its vertices lie on grid points.

Theorem 3.1 (Pick's Theorem) *[CM91] Given a grid $\mathcal{G}$ and a polygon P such that all its vertices lie on grid points, the area of P is given by the formula $A = I + \frac{B}{2} - 1$, where I and B are the number of vertices from $\mathcal{G}$ inside P and on the boundary of P respectively.*

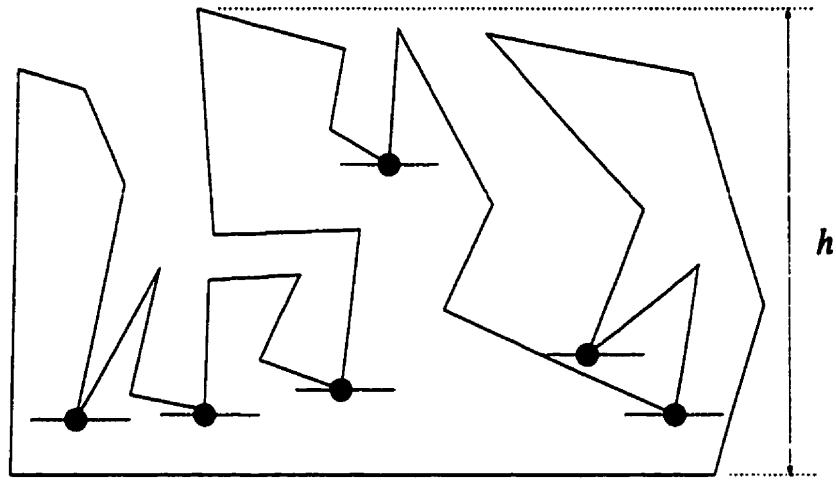

Figure 3.4: Simple polygon with 6 local minima.

Given a simple polygon over a grid, the number of squares must be known in order to compare solutions. The following two sections show how to count the number of squares from a grid that lies entirely inside a polygon.

3.2 Counting Intact Squares Inside a Convex Polygon

The squares in P form an orthogonal polygon Q. We use this in the following lemma to compute the number of squares inside a convex polygon. Let m be the number of vertices of P and h be the distance between the supporting wall and the furthest vertex of P.

Lemma 3.1 *Given a convex polygon P placed on a grid $\mathcal{G}$ (with respect to the conditions of Problem 1), we can count the number of intact squares from $\mathcal{G}$ inside P, as well as the number of squares from $\mathcal{G}$ intersecting the boundary of P in $O(m+h)$ time.*

Proof. Let l_1 and l_2 be two consecutive horizontal lines on the grid. If these two lines intersect P, we can compute the number of intact squares inside P and

lying between l_1 and l_2 in $O(1)$ time. This can be done using the intersection points between the boundary of P and l_1 and l_2. We only need to find the difference between x-coordinates (truncated and rounded) of those points. Figure 3.5 shows an example. It takes $O(h)$ time to find all intersection points between ∂P and horizontal lines from the grid, since P has height h. This gives a method running in $O(m+h)$ time. ∎

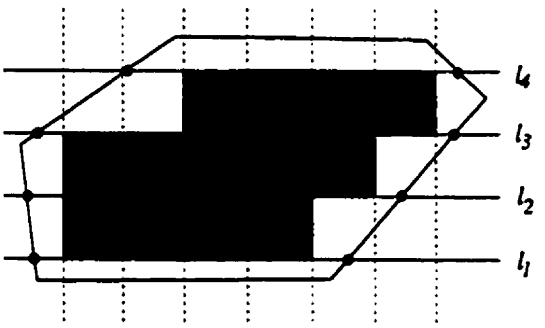

Figure 3.5: Counting the number of squares between consecutive lines. There are 4 intact squares between l_1 and l_2, as well as between l_3 and l_4. There are 5 squares between l_2 and l_3.

In general, counting the number of squares inside a convex polygon P can be done in $O(m)$ time where m is the number of vertices. P can be partitioned into axis-parallel rectangles and right triangles, as shown in Figure 3.6. The total number of pieces is $O(m)$. Let T be a right triangle whose edges share the right vertex have length s and t. The number of squares intersected by the hypotenuse of T is at most $\lceil s \rceil + \lceil t \rceil + 1$. This leads to the following:

Lemma 3.2 *Given a convex polygon P placed on a grid $\mathcal{G}$ we can count the number of intact squares from $\mathcal{G}$ inside P in $O(m)$ time.*

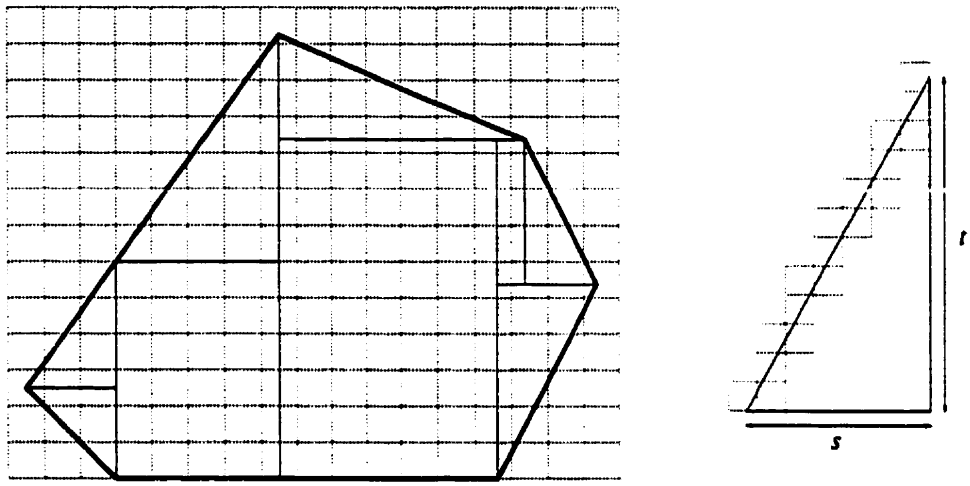

Figure 3.6: Partitioning of a convex polygon into rectangles and right triangles.

3.3 Counting Intact Squares Inside a Simple Polygon

This section shows how to find the number of intact squares from $\mathcal{G}$ inside a simple polygon P when a supporting wall W is given and all vertices lie above or on it. Let m and l be the number of vertices and local minima in P respectively. Figure 3.7 shows an instance of our counting problem where P is a simple polygon and Q is the union of all intact squares. Note that Q may be a set of disjoint polygons. But it does not cause any problem. The first step is to compute intersection between ∂P and horizontal lines from the grid, as explained in the next lemma.

Lemma 3.3 *It takes $O(m+hl)$ time to compute the set of intersection points between a simple polygon and horizontal lines of the grid.*

Proof. The algorithm is similar to the one used in Lemma 3.1. We start at the supporting wall W and walk on the edges of P towards the highest vertex in P. Let s be the starting point of the walking process which is the right endpoint of W. Two events can occur as we walk around the boundary. Either we cross a grid line or we

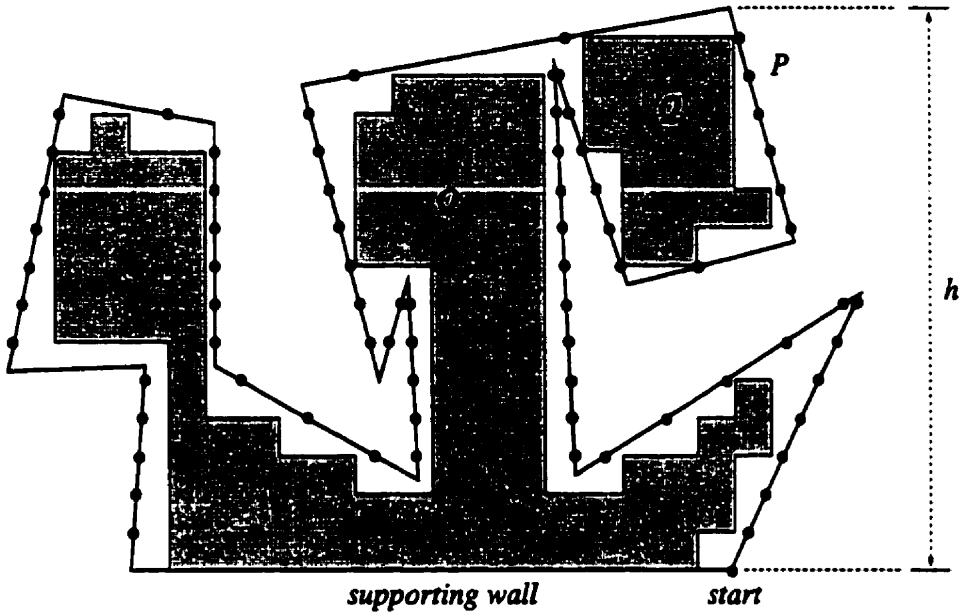

Figure 3.7: Orthogonal polygon Q inside a simple polygon P. Q is the union of all intact squares from $\mathcal{G}$ inside P. Black vertices along the boundary of P are intersection points.

come to the end of an edge. Every time we cross a horizontal grid line, we have found an intersection point. Each event costs $O(1)$ time to process. There are m edges in P and we cross at most hl grid lines. The total running time is $O(m + hl)$. ∎

In the worst case the number of local minima of a polygon is equal to the number of reflex vertices (see Figure 3.8). Using the set of intersection points between horizontal lines of $\mathcal{G}$ and the boundary of the polygon, we show how to count the number of intact squares inside P. We define a data structure for the set of those points. For each oriented edge of P, we create a doubly linked list, containing its intersection points. We now define three types of edges. An edge $e = (a, b)$ is ascending if the y-coordinate of b is greater than the y-coordinate of a, it is descending if the y-coordinate of b is less than the y-coordinate of a and horizontal otherwise. We do not need to consider horizontal edges.

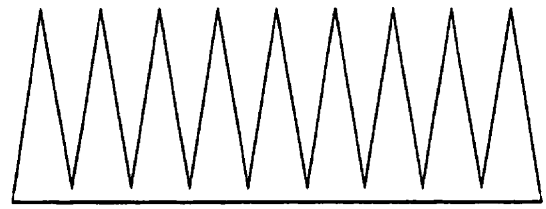

Figure 3.8: Worst simple polygon. It has 19 vertices, 8 of those are reflex vertices.

The total number of nodes in lists of ascending edges is equal to the total number of nodes in lists of descending edges. Each node takes $O(1)$ size memory because it contains the coordinates of an intersection point between ∂P and a horizontal line of the grid. It contains also a pointer to the header of the list in order to access the ID of the list and its type. Creating one list is straightforward. If we walk on an ascending edge from its endpoint with the smaller y-coordinate, then we can append into its list in $O(1)$ time the intersection points in order in which we hit them. Thus, a node n_i is adjacent to another node n_{i+1} if and only if the y-coordinates of their intersection points differ by 1. We can create a doubly linked list for an descending edge, using a similar strategy. Again, two nodes in a list of that kind of edge are adjacent if the y-coordinates of their intersection points differ by 1. As shown in Figure 3.9, Edges e_1 and e_4 are ascending edges and e_3 and e_5 are descending edges. Each edge has a list of intersection points.

We add a pointer from a node in ascending list to a node from a descending list if their y-coordinates are equal. Since we may have many nodes with the same y-coordinates we add the following rule. A node a from an ascending list has a pointer to a node d from a descending list if the line segment passing through their corresponding intersection points lie entirely inside P. Figure 3.9 shows such an example. Intersection points b, h, u and t have same y-coordinate. We add a pointer from b to t but not from b to t because $\overline{bv}$ intersects the exterior of P. Conversely, a pointer is added from h to b. Building this data structure takes $O(m + hl)$ time because a simple polygon has at most $O(hl)$ intersection points between ∂P and horizontal lines from the grid.

Using this data structure, we can count the number of intact squares inside P. There are two squares between cd and gf but there is no square between rs and wv. Since c and d are intersection points from list e_1 and g and f are also intersection points from list e_5, we have two squares (by truncating the x-coordinate of c (or d) and rounding the x-coordinate of g (or f)). Thus, we can count the number of squares between two consecutive horizontal grid lines in $O(k)$ time using the previous data structure, where k is the number of intersection points to consider. Figure 3.10 shows an example.

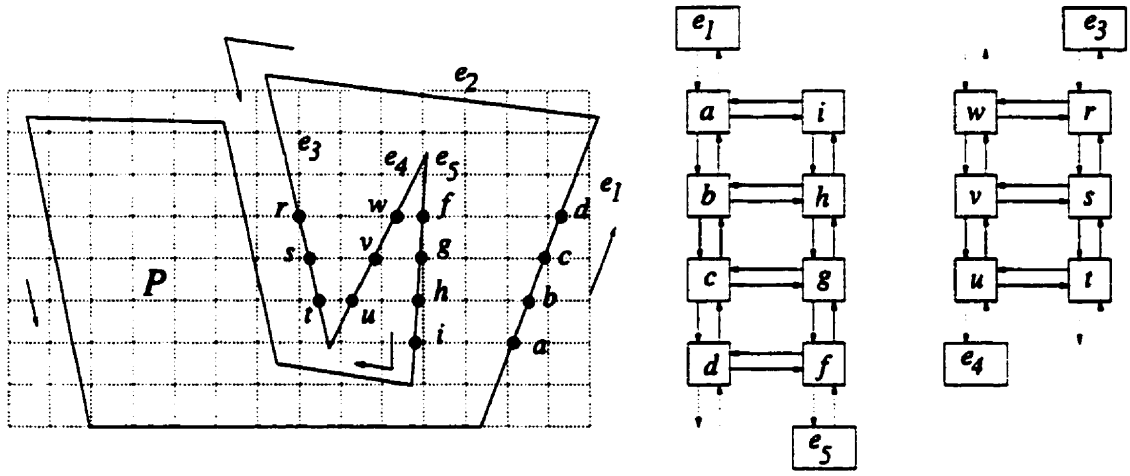

Figure 3.9: Example of doubly linked lists used to store intersection between horizontal lines of $\mathcal{G}$ and the edges of the polygon. Edges e_1, e_2 and e_4 are ascending edges and e_3 and e_5 are descending edges.

Lemma 3.4 *It takes $O(m + hl)$ time to count all intact squares inside a simple polygon with m edges, l local minima and height h.*

Proof. The process scans linked lists defined above for a simple polygon and its intersection points between its boundary and horizontal lines from the grid. At each step, we have to find the number of squares between two consecutive horizontal lines of the grid, as well as between intersection points with these lines. This can be done in $O(k)$ time, as explained above.

To count the number of intact squares between two consecutive horizontal lines of the grid, we have to compare the coordinates of four intersection points and possibly endpoints of the corresponding edges. Two of these vertices lie in an ascending list while the two others lie in a descending list. Each intersection point is involved at most two times in the whole process because each step counts squares between two consecutive horizontal lines. The algorithm stops when all lists have been visited. Thus, it takes $O(m + hl)$ time. ∎

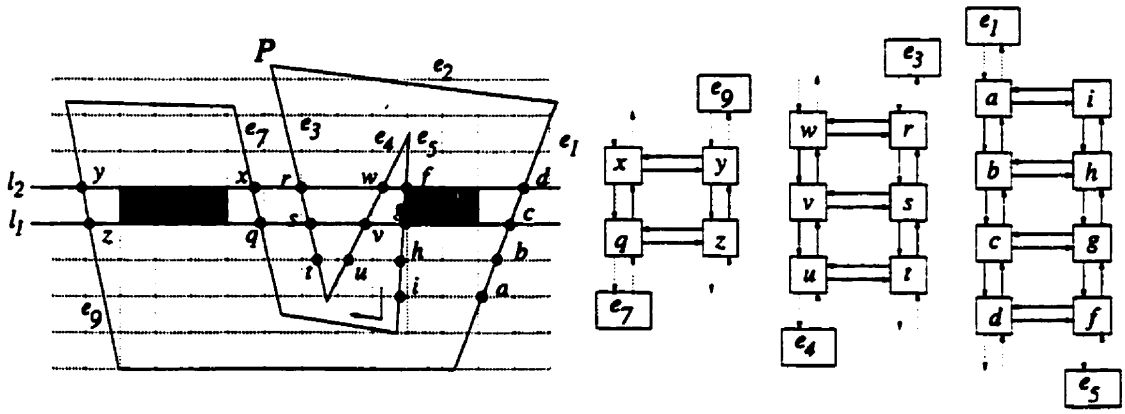

Figure 3.10: Finding the number of intact squares inside a simple polygon P using intersection points between horizontal grid lines and the boundary of P.

We have to be careful while counting the number of squares because it would be possible that we have to compare nodes belonging in 3 (and even 4) different lists. Refer to Figure 3.11. We want to count squares between l_1 and l_2, two consecutive horizontal lines from the grid. If Intersection Point d would have been in list e_3 as well as c, then we could count the number of squares using the strategy discussed earlier. However, b lies in list e_6 and the process of counting intact squares should split. First we find the number of squares between ab and qc (which is 2), and then count squares between ed and rf (which is 0).

The upper bounds from lemmas 3.3 and 3.4 are tight because there are simple polygons having $\lfloor h \rfloor$ intersection points for each edge. Since there are l local minima in a simple polygon, we may have $\Theta(hl)$ intersection points.

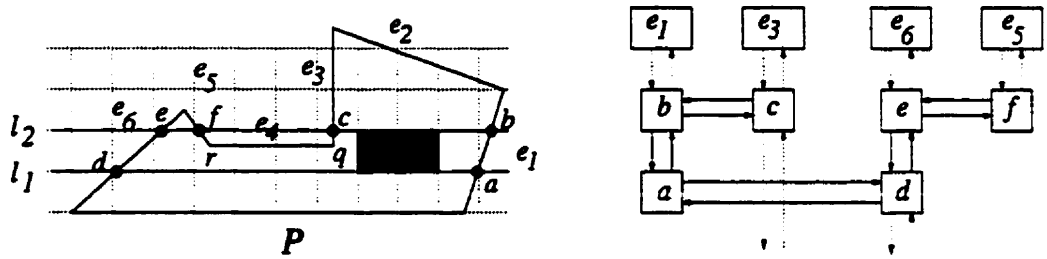

Figure 3.11: Finding the number of intact squares inside a simple polygon P using intersection points between horizontal grid lines and the boundary of P. q and r are vertices of P.

3.4 Properties of an Optimal Placement

We are now ready to characterize the optimal solution when only x-translations are allowed as well as x and y translations.

Theorem 3.2 *There exists an optimal solution to Problem 1 such that a square lying entirely inside P is in contact with ∂P, but not with its supporting wall.*

Proof. Without loss of generality, let the supporting wall W be parallel to the x-axis. Let $\tau(P)$ be an optimal placement of P on a grid. Let n be the number of intact squares lying in $\tau(P)$. If one square s has a contact with the boundary of $\tau(P)$ then the theorem is verified. Otherwise, translate $\tau(P)$ until a square is hit by $\partial\tau(P)$. One of the following two events happens. A grid point lies on $\partial\tau(P)$ or a vertex of P lies on an edge of a square. Hence, a square of the grid is in contact with the boundary of P. Such a translation exists because we can move the polygon to the right by $\epsilon > 0$ unit, where ϵ is the smallest horizontal distance between $\tau(P)$ and a grid square. ∎

The following theorem states the property of an optimal solution to the placement problem where both x and y translations are allowed.

Theorem 3.3 *There exists an optimal solution to Problem 2 where the boundary of P has at least two contacts with intact squares from $\mathcal{G}$ lying inside P.*

Proof. Let $\tau_1(P)$ be an optimal placement of P on the grid. Let n be the number of squares inside $\tau_1(P)$ If there are two edges e_1 and e_2 from P in contact with squares from then the statement of the theorem is verified.

Otherwise, translate $\tau_1(P)$ to the right (x-translation) until its boundary hits a square s_1. Let $\tau_2(P)$ be this placement of the polygon. First we take a look when an edge of P is in contact with a grid point. Let e_1 be that edge. Observe that the number of intact squares from $\mathcal{G}$ inside the polygon is still n. The second operation is a translation along Edge e_1. We translate $\tau_2(P)$ until a square s_2 is hit by an edge or a vertex of P. Hence we have now at least two contacts between the boundary of the polygon and squares from the grid.

Once the first translation have been made, the contact may be a vertex v of P lying on an edge of Square s_1. Let s_e be the edge of s_1 which is in contact with v. If it is the case, then we do a translation of the polygon from $\tau_2(P)$ along s_e such that v still lies on the vertical line passing through s_e, until a square is hit by the boundary of the polygon. Again, the number of intact squares lying inside P is still n since no square has been added or lost. Therefore there exists an optimal solution where the boundary of P has two contacts with intact squares lying inside P. ∎

As discussed in the proofs, two types of events may happen when translating the polygon:

1. An edge of P hits a grid point.
2. A vertex of P hits an edge of a grid square.

We call these critical events. During a translation of P, some critical events happen and the quality of the placement of the polygon on a grid may change. In other words, when such an event happens, the number of intact squares inside the polygon may change.

Chapter 4

Sorting Grid Points

In this chapter we solve the following problem.

Problem 4.1 *Given a line segment e with slope different from 0 on a grid, sort in linear time by their horizontal distance from e, the set of grid points lying on the same side of e whose horizontal distances are less than one.*

This is an important step in finding an optimal solution to Problem 1. Figure 4.1 shows an instance of the problem. The original line segment lying on the grid is denoted e_0.

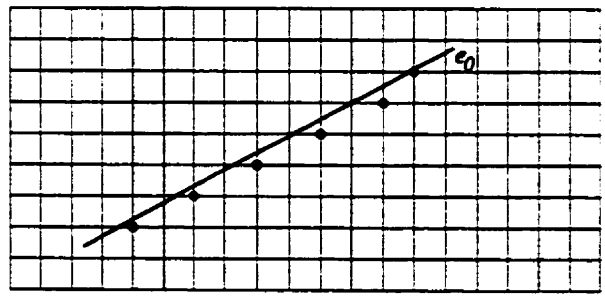

Figure 4.1: An instance of Problem 4.1. A line segment e_0 and 6 grid points to sort by their corresponding horizontal distance from e_0. All distances are less than 1.

Algorithm A1 described in Section 5.1 has an important step in which the set of its nearest grid points are sorted with respect to their horizontal distances to the

polygon. Algorithm A1 translates a polygon P to the right and updates the current solution when a grid point is hit by ∂P. Knowing the order in which the grid points are met enables us to design an efficient algorithm. The maximum distance that P is translated is 1 unit since any translation by t units can be represented as $t = k + \epsilon$, where k is an integer and $0 \leq \epsilon < 1$. Because these points lie on the grid, we will be able to perform the sorting in $O(n \log m)$ time, where m is the number of vertices in P and n is the number of nearest grid points. Our algorithm explained in the next section is used instead of well known algorithms taking $\Theta(n \log n)$ time [CLR90, Man89]. In fact, if the number of edges in P is greater than n then it would be more appropriate to use MergeSort or QuickSort. However, the geometry of the problem allows us for a linear time sorting algorithm, with respect to a line segment.

4.1 Main Sorting Algorithm

In this section we present the algorithm which sorts the set of nearest grid points in linear time for a line segment. The method is based on results discussed in the following sections. The original edge is labeled e_0. We first apply a horizontal translation such that e_0 hits a grid point. This new segment is labeled e_1. The second step is rotating e_1 around this first grid point until a second grid point is hit. The resulting edge is called e_2. Doing so does not change the sorted order of the grid points, as proven in Section 4.2. Figure 4.2 shows an example. This new edge e_2 is used to sort the set of grid points. We apply Lemma 4.2 to sort grid points lying in triangles 1 and 2, in linear time. This lemma explains how to sort grid points when edge e_2 passes through exactly two grid points (at its endpoints). We need Lemma 4.3 if there are more than two triangles. This lemma shows how to sort grid points lying in a set of right triangles, as the ones shown in the figure.

Figure 4.2 shows an example where O is the grid point having the smallest y-coordinate, among all points to sort. But what if there are other grid points to sort below O? Then we may split e_1 into two smaller line segments sharing one endpoint, which is O. The next steps of the algorithm apply Lemmas 4.2 and 4.3 on those smaller line segments. Figure 4.3 shows the main steps of the method.

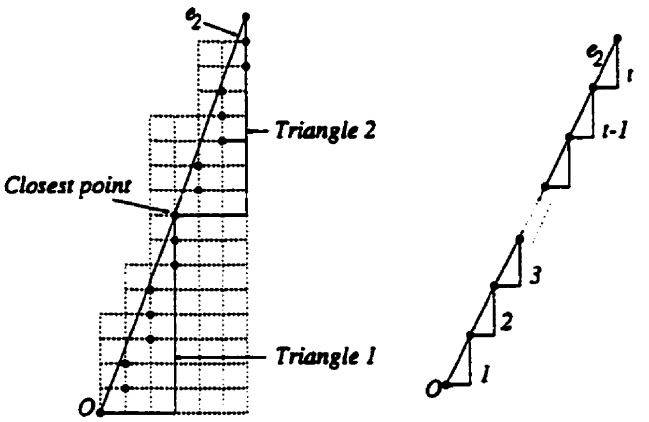

Figure 4.2: e_2 and the set of grid points (left) and its set of triangles (right).

Algorithm Sorting-Grid-Points(e_0)

Input: Line segment e_0, lying on Grid $\mathcal{G}$.

Output: A sequence of grid points sorted by their horizontal distances to e_0.

Each grid point lies at less than 1 unit from e_0.

BEGIN

$e_1 \leftarrow$ Translate e_0 to the right until a grid point O is hit.

Let A and B be sets of closest grid points from e_1 above and below O respectively.

FOR set A (and later for B)

$e_2 \leftarrow$ Rotate e_1 clockwise around O until a point from A lies on e_1.

For A, e_2 is the rotated part of e_1 above O (below O in the case of B).

$T_A \leftarrow$ Set of triangles.

$Z_A \leftarrow$ trapezoid (not complete last triangle). *As shown in Figure 4.3.*

For a triangle in T_A, sort its grid points using Lemma 4.2 in linear time.

For other triangles in T_A, assign the corresponding rank to each grid point.

Assign ranks to grid points in Z_A, using the ranks from the previous step.

Apply bottom-up argument to merge grid points from the set of triangles.

$L_A \leftarrow$ Merge sorted lists T_A and Z_A. *As shown in Figure 4.3.*

Merge sorted grid points from A and B.

END.

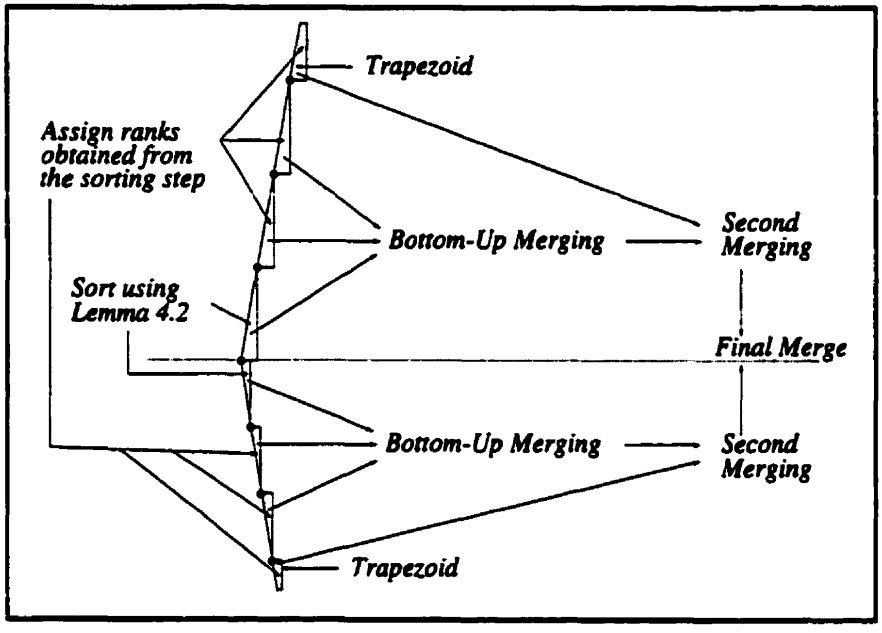

Figure 4.3: Steps of the sorting algorithm. $O(1)$ number of merging steps. This leads to a linear time algorithm.

4.2 Results

In this section we prove the basic tools used by our sorting algorithm. The following lemma explains how a small rotation of a line segment over the grid does not change the ranks of the set of grid points. Let $O = (0, 0)$ be the first grid point hit by e_0 as it is translated to the right. A translation does not change the rank of any grid point since we are sorting by horizontal distance to e_0. Let $v \in S$ be the first grid point being hit by e_1 while it rotates clockwise around O. Let e_2 be the rotated copy of e_1 around O. The angle of rotation is determined by v such that e_2 passes through v. The x and y coordinates of v are (v_x, v_y). We denote by $d_h(q, e)$ the horizontal distance between a point q and a line segment e.

The following lemma states that the rank of grid points lying below v does not change. Refer to Figure 4.4.

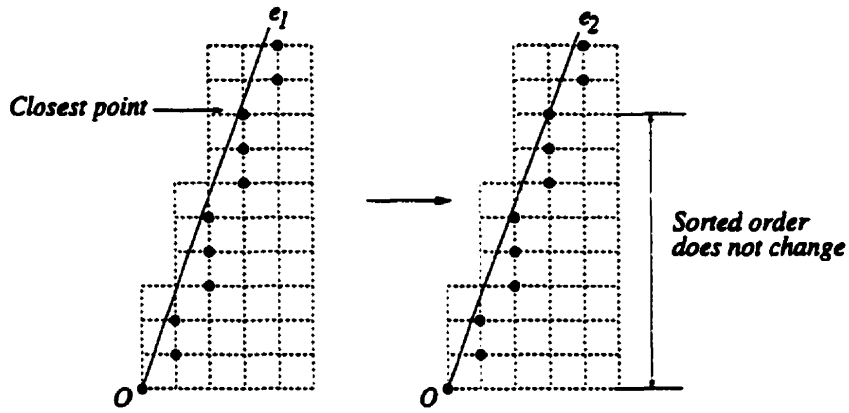

Figure 4.4: Rotating e_1 to the closest point v does not change the sorted order of the grid points lying below v and above O.

Lemma 4.1 *Rotating e_1 around O until a grid point $v \in S$ is on e_1 does not change the rank for all $u \in S$ where $0 < u_y < v_y$.*

Proof. (By contradiction) Suppose that the ranks of two points change. In other words there are two grid points p and q in S such that $d_h(p, e_1) < d_h(q, e_1)$ and $d_h(p, e_2) > d_h(q, e_2)$ for $p_y < q_y < v_y$. Let d be the euclidean distance between p and q. Let l_1 and l_2 be two lines containing segments e_1 and e_2 respectively. Let L be a line passing through p and q. Hence, L intersects both l_1 and l_2, because the horizontal distances from those lines to vertices p and q are different. Let s and t be those intersection points respectively. We have $s_y < p_y$ because the slope of L is smaller than the slope of e_1. We also have $t_y > q_y$ because the slope of L is greater than the slope of e_2, as shown in Figure 4.5.

Now, let L' be a line parallel to L and passing through O. We know that the slope of e_1 is greater than the slope of L. Once the rotation has been made, the slope of that line segment becomes smaller than the slope of L'. But the rotation of e_1 could not go past L' because there must be a point q' on L' at distance d from O. Because O is a grid point, q' should have been chosen as the closest point. This contradicts the fact that v was the closest grid point. ∎

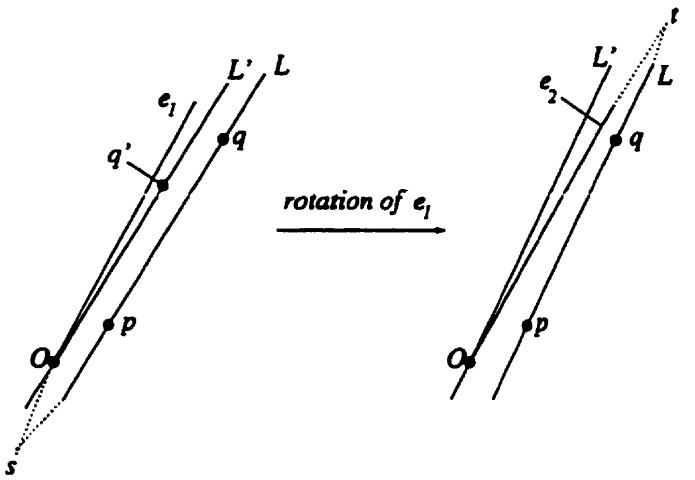

Figure 4.5: Configuration of the proof of Lemma 4.1.

The following lemma and its proof show how to sort the set of nearest grid points lying below v and above O, according to the notation used in the previous proof. In fact, it sorts the nearest grid points lying in a right triangle with vertices O, v and $(v_x, 0)$, (see Figure 4.6). There may be a subset of grid points outside the triangle defined above. However, we will explain in Section 4.3 how to deal with those points. Let S be a set of n grid points lying on the same side of e_2 and having their horizontal distance from e_2 less than 1.

Lemma 4.2 (Sorting in a right triangle) *Given a set S of n grid points and a line segment e_2 passing through exactly two grid points which are also its endpoints, we can sort the points in S with respect to their horizontal distances with e_2 in $O(n)$ time.*

Proof. Let O and v be grid points that are the endpoints of e_2. All points from S lie in the triangle T with vertices O, v and $(v_x, 0)$. The only grid points that lie on e_2 are O and v. See Figure 4.6 (A).

We know that all points in S lie in T. We now apply a scaling operation to this triangle. Each square is subdivided into $v_y \times v_y$ smaller squares. We divide by v_y since the number of grid points to sort depends on the height of the triangle. This implies

that the intersections of horizontal lines lie on grid points of the new grid. This is due to the slope of e_2 which is v_y/v_x and the coordinates of the grid points to sort are multiples of v_y in the finer grid. Therefore, horizontal distances from $u \in S$ to l are now integers in the range $[1, v_y - 1]$. There are $v_y - 1$ integers to sort. Since the rank of each point is its distance to e_2 in the finer grid, we can compute each rank in $O(1)$ time using the intersection points between the hypotenuse and horizontal lines. Observe that all points in S have different distances: if two grid points p, $q \in S$ have the same horizontal distance from e_2, then it means that e_2 is passing through at least three grid points. Therefore, we sort in $O(n)$ time, because there are n points.

■

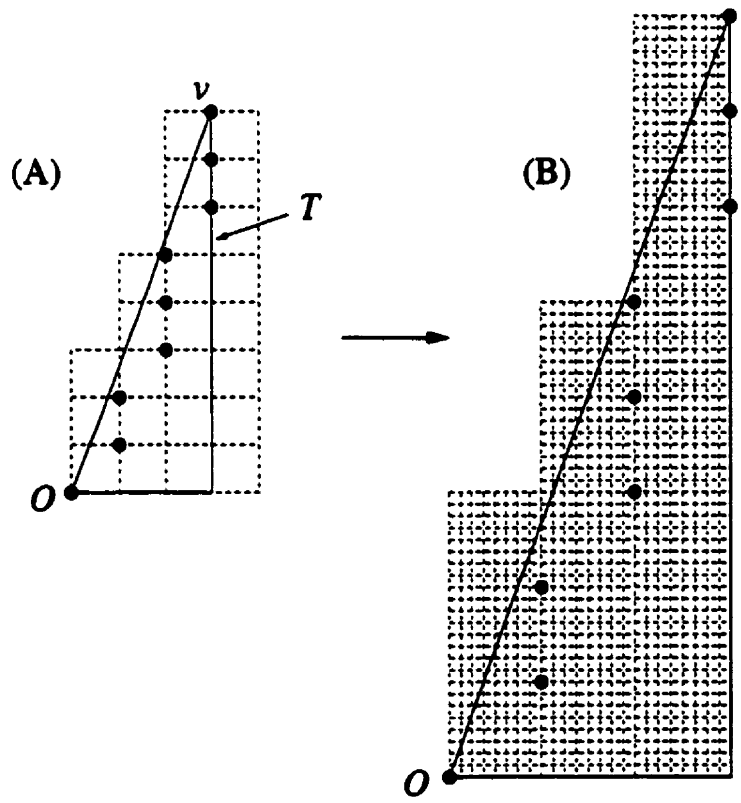

Figure 4.6: Example of constructing a finer grid from another. In (A) 7 grid points to sort. Creating a finer grid (8×8) from the triangle O, v and $(v_x, 0)$ makes distances between each grid point and the line integers.

The previous lemma only applies to line segments with slope different from 0 and 1. If the slope is equal to 0 then there is nothing to do because the set of nearest grid points is empty. If the slope is equal to 1 then all grid points are equally distant from the line segment (the distance may be equal to 0). Therefore, we only need to compute one distance. This also applies to vertical line segments. Finally, if the difference between the x-coordinates of the rightmost and leftmost grid points is at most 1, then the points are already in sorted order.

4.3 Bottom-Up Argument

An important step of our sorting algorithm is bottom-up merging, because we may encounter many triangles once the line segment has been rotated. These triangles have equal internal angles and dimensions. Also, the set of grid points to sort in each triangle is the same with respect to their coordinates and the hypotenuse of the corresponding triangle. Thus, if we know the sorted order of a subset of grid points with respect to a triangle, then we know the sorted order for grid points with respect to their triangles. In other words, the configuration is symmetric and we can assign to each grid point a rank.

The problem that we solve now is merging the ranks given by Lemma 4.2 for each grid point. However, we have to merge these points with respect to their distances from the original edge e_0, not e_2! Since the final order does not change from e_0 to e_1, we use e_1. We know that both e_1 and e_2 pass through O. Let p be a grid point from S. Let p_i^j be the grid point with rank i coming from triangle j. The triangle with lowest y-coordinate takes index $j = 1$. Assume that all triangles lie above O.

Let $b - 1$ be the number of points to sort in each triangle. If there were only one triangle, then p_i would take place at index i at the end of the sorting process. In the final sorted list, the rank of p_i^j is smaller than the rank of p_i^{j+1}, (even if their distances from e_2 are equal) because the latter lies in a triangle further from O than the former point. This is due to the fact that the rank has been found using e_2 and this edge is a rotated copy of e_1. In other words, $d_h(p_i^j, e_1) < d_h(p_i^{j+1}, e_1)$. We know that p_i^j lies before p_{i+1}^{j+1} and so does p_i^{j+1} versus p_{i+2}^j. Refer to Figure 4.7. If you rotate back e_2

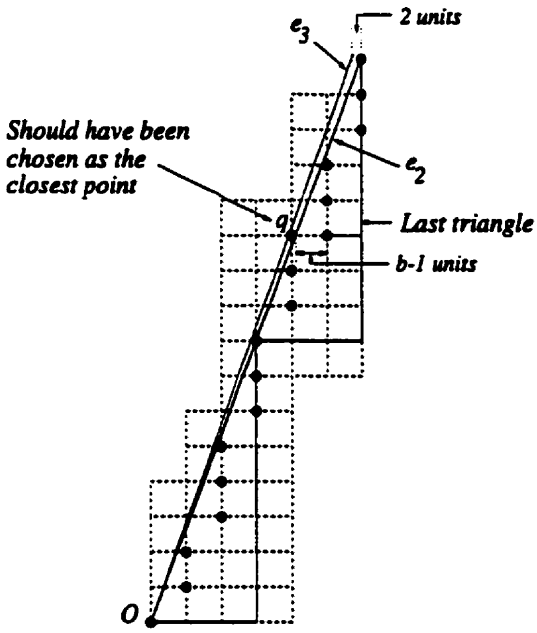

Figure 4.7: Rotating back e_2 too far may modify the set of nearest grid points.

such that its horizontal distance from its original endpoint lies at distance 2 then a grid point q changes side of e_2 (becoming e_3 after the rotation). Grid point q is on the left of e_2 but is on the right of e_3. This is due to the fact that there is a grid point in S that lies at distance $b-1$ units from e_2 if you partition each square of the grid into $b \times b$ smaller squares. Hence q should have been the closest point.

But what can we say about p_{i+1}^j and p_i^{j+1}? It is not obvious that p_i^{j+1} lies before p_{i+1}^j in the sorted list. While we rotate e_1 towards S, the further we are from O, the greater the horizontal distances from e_1 become. The following lemma shows that p_i^j lies before p_{i+1}^k for any $k \neq j$.

Lemma 4.3 *Let c be the number of triangles once e_1 has been rotated. Each triangle is labeled by $1 \leq j \leq c$ and has $b-1$ points to sort. The rank of p_i^j is smaller than the rank of p_{i+1}^k for any $1 \leq j, k \leq c$ and $1 \leq i \leq b-2$.*

Proof. Let S be the set of grid points to sort by their horizontal distances. Let e_2 be the line segment resulting from a rotation applied to e_1 around the origin O to the

closest grid point. We need to prove that if we rotate e_2 back to e_1 the sorted order of the grid points from S does not change.

First, observe that if we rotate back e_2 by an angle $\epsilon > 0$ then the sorted order does not change. We have a finite number of triangles and grid points, and ϵ may be as small as we want. The rotation of e_2 does not imply that e_2 is passing through another grid point because ϵ is very small. However, the angle needed to get e_1 again may be much bigger than ϵ. Now, suppose that after the rotation of e_2 two grid points swap their ranks in the final order. Let a and b be these two points, where $d_h(a, e_2) < d_h(b, e_2)$. The argument that leads to a contradiction is the same as the one used in the proof of Lemma 4.1.

Let e_δ be a rotated copy of e_2 around O, where $d_h(a, e_\delta) > d_h(b, e_\delta)$ and δ is the angle between e_δ and e_2. Let L be the line passing through a and b and let L' be the line passing through O and parallel to L. There exists an angle $\gamma > 0$ such that e_γ is parallel to L. This means that $d_h(a, e_\gamma) = d_h(b, e_\gamma)$. Thus, there exists a grid point b' lying on L', where the line segment $\overline{Ob'}$ is a translated copy of $\overline{ab}$. Since we assumed that $d_h(a, e_\delta) > d_h(b, e_\delta)$, the rotation process brings e_δ further than L'. Observe that b' lies on the left (right) of e_2 and after the rotation, b' lies at the right (left) of e_δ. Because b' lies on a different side of the rotated line segment, b' should have been chosen as the closest grid point. This contradicts the fact that we chose the closest grid point before applying a rotation to e_1. ∎

Hence the results for the final sorted order are the following:

- p_i^j lies before p_{i+1}^j;
- p_i^j lies before p_i^{j+1};
- p_i^j lies before p_{i+1}^k for any $k \neq j$.

Therefore, the merging algorithm for grid points lying in a set of triangles is simple. Let t be the number of right triangles produced once the rotation step is done. The first $t + 1$ grid points to lie in the sorted list are the ones lying at distance 0 from e_2 (these are vertices of the triangles, as shown in Figure 4.2). The next t points are currently lying at $1/b$ unit from e_2, in the finer grid as explained in Lemma 4.2. And we keep going on until all grid points are placed in the sorted list. This completes the explanation on how to merge grid points lying in triangles in linear time. Previous

results lead to the following:

Theorem 4.1 *Given a line segment e lying on a grid, its closest grid points can be sorted in linear time with respect to their horizontal distances to e.*

Merging Sorted Lists

In Section 5.1, Algorithm A1 has to deal with a set of line segments. Hence, we have to merge sorted lists created by the previous sorting algorithm. The next result shows how to sort m sorted lists where the total number of elements in the lists is n. However, there is no upper bound on the number of elements in a list.

Theorem 4.2 *Given m sorted lists where the total number of elements is n then we can merge them in $O(n \lg m)$ time.*

Proof. Let H be a heap of size m. We first insert in H the smaller element of each list. For each element, we have to keep in memory where it came from (ID of the list). Removing and adding an element in H takes $O(\log m)$ time.

Removing the first element of H, say h_1 ensures us that it is the smallest element available. No other element from the lists is smaller. When an element is removed from H, a new one is added to it. This element comes from the same list as h_1. If the corresponding list is empty, then we can continue the process and H contains only $m-1$ elements. At the end of the process, the lists and the heap are empty. Therefore, it takes $O(n \log m)$ time to sort n elements because we have to perform exactly one insertion and two deletion operations for each element. (One deletion from its list and one deletion from H.) ∎

Chapter 5

Placement Algorithms

This chapter shows how to solve problems of finding an optimal placement of a polygon on a grid efficiently, depending on the operations permitted. It is the core of the thesis. First, Section 5.1 shows how to solve Problem 1 (see page 13), where only x-translations are allowed. An algorithm solving Problem 2 (defined on page 59) is presented in Section 5.2.

5.1 Algorithm A1

This section presents an algorithm solving Problem 1, where only x-translations are allowed. The algorithm works for convex and simple polygons and takes $O(m+h\log m)$ and $O\left((m+hl)\log m\right)$ time respectively. Where m is the number of vertices in the polygon and h its height. In the nonconvex case, the complexity of the algorithm depends also on the number of vertices being local minima (y-coordinates), denoted by $l \geq 1$. If the polygon does not have any local minimum, the complexity of our algorithm is $O((m+h)\log m)$. First, we define the monotonicity of a polygonal line and a polygon.

Definition 5.1 (Monotone chain) *A polygonal chain C is strictly monotone with respect to a line L if the intersection of C with every line orthogonal to L is either a point or empty.*

Definition 5.2 (Monotone polygon) *A polygon Q is monotone with respect to a line L if the boundary of Q can be partitioned into two polygonal chains where each one is monotone with respect to Line L.*

We define a special set of grid points lying near the boundary of a polygon P.

Definition 5.3 (Nearest neighbor) *A grid point p is a nearest neighbor of a line segment e if p lies to the right of e and its horizontal distance to e is less than 1.*

5.1.1 Main Idea of Algorithm A1

Algorithm A1 is based on the fact that each square from the grid has one of the following properties, as shown in Figure 5.1:

1. A square may be entirely inside P
2. A square may intersect ∂P
3. A square may be entirely outside P

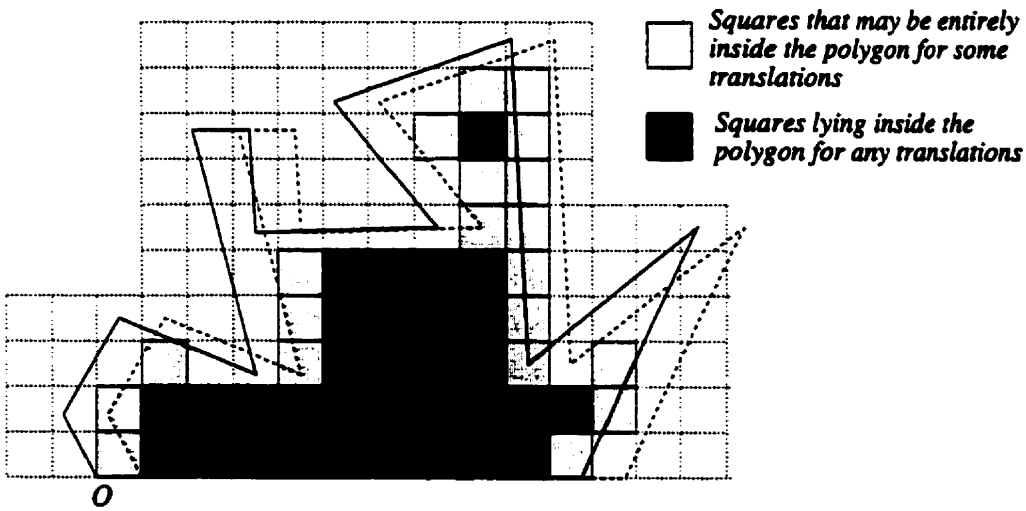

Figure 5.1: Simple polygon and a translated copy (dashed edges) on a grid.

Refer to Figure 5.2. We want to know when P will hit a grid point if we translate it to the right only. The set of nearest neighbors is shown in the figure.

If P is translated to the right by less than 1 unit, then grid points hit during the motion can only be the nearest neighbors. When a nearest neighbor is hit, some square(s) from the grid may be lost or new square(s) may "enter" inside P. It is possible that more than one square enters completely inside P for a given translation. We restrict translations to 1 unit because any translation $t = \lfloor t \rfloor + \epsilon \geq 1$ is equivalent to a translation of $\epsilon \geq 0$, due to the periodicity of the grid. Given a placement of P on the grid, translating P to the right by exactly k units, where k is an integer, results in an equivalent placement of P.

Our main goal is to translate the polygon efficiently one unit in order to find the placement containing the maximum number of squares inside P. The following is the pseudo-code of the algorithm finding an optimal solution:

Algorithm A1(P)
Input: A simple polygon P having an edge parallel to the x-axis.
Output: Solution to Problem 1. Assume that the tile size is 1.
BEGIN
    $e_k \leftarrow$ horizontal edge.
    Let the origin O of P be the leftmost endpoint of e_k.
    $N_1 \leftarrow$ Compute the set of nearest neighbors of P.
    $N_2 \leftarrow$ Sort the neighbors from N_1 by their right horizontal distances from ∂P.
    $V_r \leftarrow$ Set of reflex vertices of P that may be in contact with a grid square.
    $E \leftarrow$ Merge N_2 and V_r, with respect to their horizontal distances.
    counter $\leftarrow$ Number of intact squares from $\mathcal{G}$ inside P.
    Initialize *bestsolution* to *counter*.
    Initialize *placement* to O.
    WHILE E is not empty DO
        Update *counter*.
        IF *counter* > *bestsolution* THEN
            Update *bestsolution* and *placement*.
    Return an optimal placement.
END.

Where *placement* is a point on the plane, corresponding to the origin of P. The variables *counter* and *bestsolution* are integers. They keep track of the current number of squares inside P and the biggest value that *counter* had respectively. N_1 contains the set of nearest neighbors. N_2 is an array containing the sorted list of neighbors contained in N_1. The sorting method has been discussed in Chapter 4. V_r contains the set of reflex vertices (if any) that can enter inside a grid square or exit from a square. This is discussed in Section 5.1.4. E is an array containing the set of nearest neighbors and reflex vertices of P that may change the current solution. In other words, E is a list of critical events. In fact, the algorithm visits all critical events, described in Theorem 3.2.

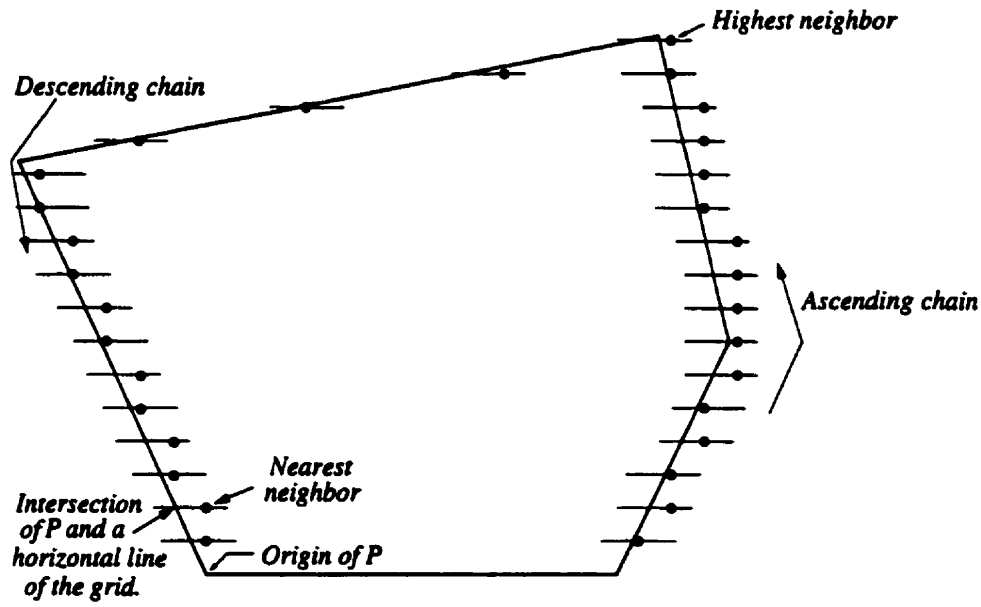

Figure 5.2: Set of nearest neighbors of a convex polygon.

The next two sections show how to compute the set of nearest neighbors for convex and simple polygons respectively. In Section 5.1.4 we discuss another type of event that occurs when the polygon is translated to the right. Section 5.1.5 shows how to determine efficiently whether or not a neighbor found in Sections 5.1.2 and 5.1.3 can be supporting vertices. Finally, We explain in Section 5.1.6 how to find an optimal

solution using the set of critical events. Section 5.1.7 discusses the worst case analysis of Algorithm A1. The correctness of the algorithm is proved in Section 5.1.6.

5.1.2 Computing the Set of Nearest Neighbors for Convex Polygons

Let P be a convex polygon of height h with its vertices in counter clockwise order. The process scans the edges of P and finds for any intersection between an edge and a horizontal line of the grid $\mathcal{G}$, the closest grid point inside or outside P. We omit horizontal edges of P.

Given a line segment e over $\mathcal{G}$, we can determine the number of grid points that are nearest neighbors of e in $O(1)$ time. However, $O(k)$ time is necessary to enumerate them, where k is the number of neighbors of e. A way to achieve this is to compute intersections between e and all horizontal lines from $\mathcal{G}$. e does not have any nearest neighbor if it is horizontal and does not intersect any horizontal lines of $\mathcal{G}$. However, if e lies on a horizontal line, then it has two neighbors which are the closest grid points to its endpoints. Therefore, the total number of nearest neighbors in the worst case is $2\lfloor h \rfloor$ when P is convex.

Refer to Figures 5.3 and 3.9. We now explain how to compute the set of nearest neighbors and how we store these. The concept is similar to the one explained in Section 3.3 about counting intact squares inside a convex polygon. Two doubly linked lists are used, labeled A and D, for ascending and descending chain respectively. One list is created for both types of chain. Since there are $2\lfloor h \rfloor$ nearest neighbors, both lists have equal size.

We also store in each node the horizontal distance from the grid point to the polygon. The neighbors are inserted in the linked list in the order they appear on the edges of P, from the supporting wall to the highest point of P, on both chains. Since P is convex, the neighbors are sorted by their y-coordinates. For each node A_i in A we add a pointer to the node D_j in D corresponding to the neighbor on the same horizontal line, and also from D_j to A_i.

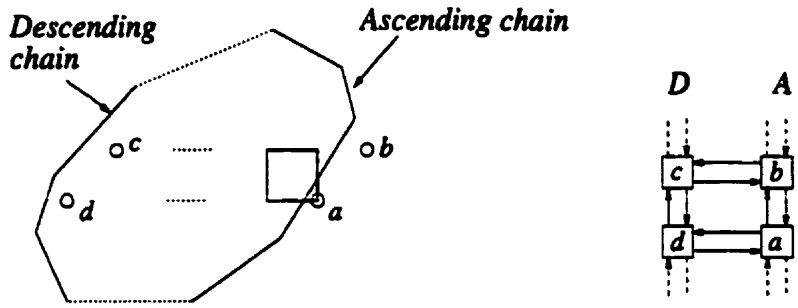

Figure 5.3: The grid points a, b, c, d are nearest neighbors of a convex polygon. Right: part of the data structure in which nearest neighbors have been stored.

5.1.3 Computing the Set of Nearest Neighbors for Nonconvex Polygons

Let P be a nonconvex polygon with height h and l local minima. The total number of nearest neighbors in the worst case is $2(l+1)\lfloor h \rfloor \in O(hl)$. This is due to the fact that each monotone chain may have at most $\lfloor h \rfloor$ neighbors.

As explained in the previous section, given an edge of P we can compute in $O(k)$ time the set of k nearest neighbors for that edge. We visit the edges of P in counter clockwise order, from the supporting wall. Every time a neighbor is found, it is added to the data structure as explained previously. Refer to Figures 5.4 and 5.5.

Each neighbor of P lying outside the polygon is matched with a neighbor lying inside P. In other words, we create pairs of neighbors where one of them is from an ascending chain and the other one from a descending chain. Let d and g be two nearest neighbors lying outside and inside P respectively, as shown in Figure 5.4. These neighbors are matched together because they lie on the same horizontal line, and g is the first neighbor inside P encountered while we walk on that line, to the left from Neighbor d. And there is no other neighbor lying on the line segment $\overline{dg}$.

Let e and f be two neighbors computed from different types of chains (ascending and descending). Let e' and f' be the intersection points between ∂P and the horizontal line passing through $\overline{ef}$. e and f are nearest neighbors for e' and f' respectively. We say that e and f form a pair if the line segment $\overline{e'f'}$ has length 0

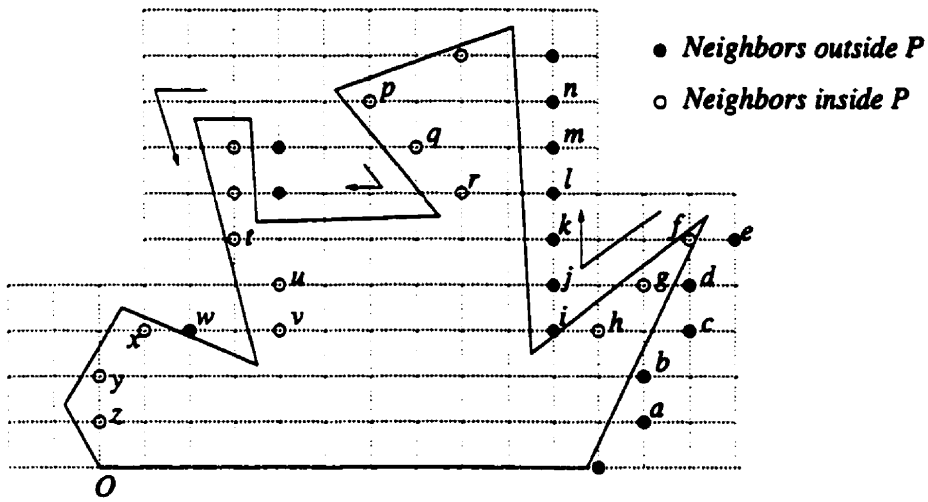

Figure 5.4: A simple polygon and its nearest neighbors.

or does not intersect edges of P, except at its endpoints. Our main goal is to put these neighbors in doubly linked lists, as shown in Figure 5.5. There is a list for each monotone chain of P. Each node contains a neighbor of P and also a pointer to its matched neighbor from a different linked list. This data structure gives us fast access to nearest neighbors and is also used in the next sections.

5.1.4 Other Events for Simple Polygons

When the polygon is simple, another type of event may change the number of intact squares inside it. As shown in Figure 5.6, a vertex of P may intersect a square while P is moving to the right. And these cases shown in the figure are not handled by the nearest neighbors.

For each square that is intersected by reflex vertices, we have to find out when it will be lost and/or added inside the polygon. Computing the set of translations takes $O(r)$ time where r is the number of reflex vertices in P. Not all reflex vertices of P are involved. In fact only those being local minima/maxima on the x-coordinates may create a an event. Fortunately, the total number of squares to visit in order to find an optimal solution to Problem 1 is still $O(hl)$. Even if r is much greater than l.

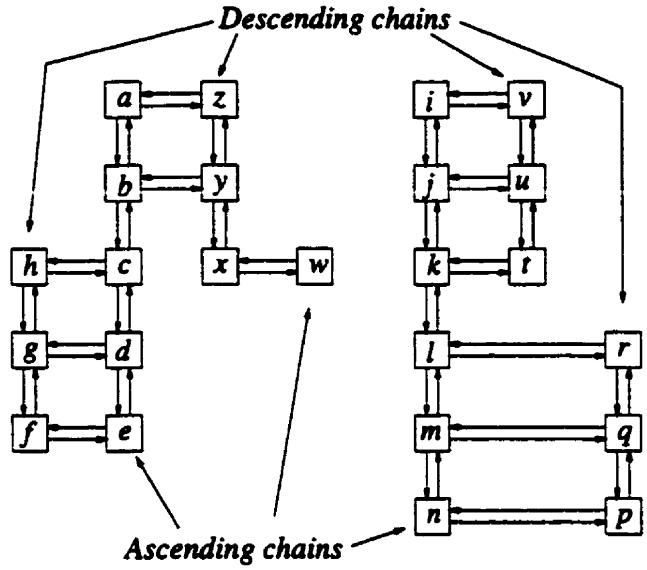

Figure 5.5: Doubly linked lists containing the set of nearest neighbors from Figure 5.4.

Once the horizontal distances from those vertices to the grid squares are known, we can sort these in $O(r \log r)$ time. The next step is merging these events with the list of nearest neighbors, in sorted order. This final list contains the set of critical events that will be visited by Algorithm A1. This operation is explained in Section 5.1.6.

The computation of these critical events first verifies if a vertex of P is a local minima or maxima on the x-coordinates. This takes $O(m)$ time. In order to determine if such a vertex is truly a critical event, we have to make sure that it can support a grid square. We only have to use the data structures containing the set of nearest neighbors, as well as other vertices of P. The next section shows how to determine if a neighbor of P can be a supporting vertex. Algorithms presented in that section can be used to determine if a reflex vertex of P lies on the boundary of a square lying entirely inside P. And thus representing a critical event.

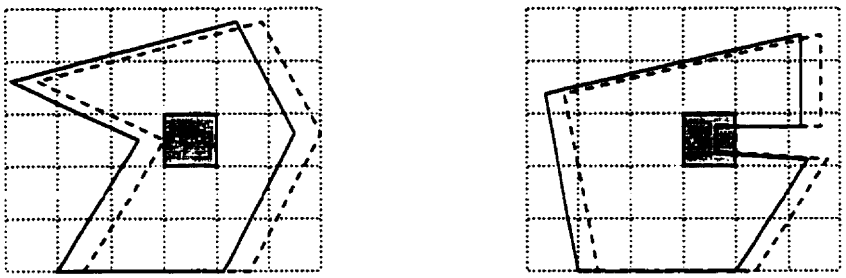

Figure 5.6: A translation to the right brings a vertex of the polygon inside/outside a grid square.

5.1.5 Determining if Neighbors are Supporting

Hitting a grid point while translating the polygon to the right does not mean that a new square is added inside it. It is possible that a neighbor is not a supporting vertex. Here we describe all possible cases and show how we can find efficiently not significant neighbors from the lists. In other words, we have to verify if each nearest neighbor can support a square which has to lie entirely inside P. The following are the cases that may be encountered, as shown in Figure 5.7:

1. A grid point hitting an ascending chain brings a new square inside P. It also means that the solution is better if we did not lose other square(s).

2. A grid point hitting a descending chain keeps a square inside P, and may be lost in a future translation (because that square shares at least one vertex with ∂P).

3. A grid point hitting an ascending chain does not put a new square inside P. This means that this grid point is not a supporting vertex.

4. A grid point hitting a descending chain and the corresponding squares does not lie inside P. That grid point is not a supporting vertex.

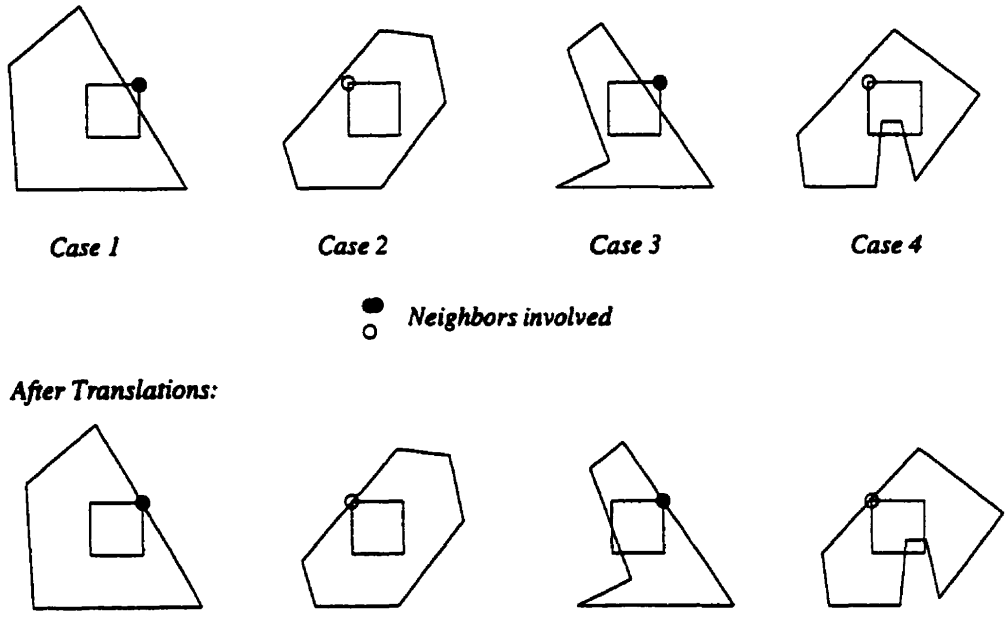

Figure 5.7: Example of cases encountered during the translation to the right of a polygon. Black and white grid points are nearest neighbors of the polygons lying outside and inside them respectively.

Figure 5.7 also shows translated polygons. The following events happened to the squares:

Case 1: The square lies inside the polygon.

Case 2: The square lies inside the polygon, but will be lost in the next translation.

Case 3: The square cannot be inside the polygon, for any small translation to the right.

Case 4: The square does not lie inside the polygon.

We have to remove the neighbors belonging to Cases 3 and 4 because they won't become supporting vertices. By Theorem 3.2 we only have to look at all possible supporting vertices and other events explained in Section 5.1.4 to find an optimal placement. In the next paragraphs we explain in more detail these cases.

It is possible that a grid point appears in both linked lists A and D. Figure 5.2 shows such an example. At the highest horizontal line from $\mathcal{G}$ intersected by P, there

is only one neighbor. But that grid point is a nearest neighbor for both chains because there exist two translations of P, less than one unit, such that this neighbor lies on P. Therefore, that grid point must be removed from both lists. We split the discussion for convex and simple polygons.

5.1.5.1 Convex Polygons

We first show how to find neighbors that cannot be supporting vertices when the polygons are convex. The next section deals with simple polygons. Convex polygons are easier to handle because there are at most two neighbors at each horizontal line intersecting them. Let P be a convex polygon. We examine Case 1, defined in the previous section.

Case 1

Let d be a nearest neighbor of P and lying outside it. Let $abcd$ be its corresponding square. Without loss of generality, assume that d is the lower right corner of $abcd$. Let $d_h(d, P)$ be the horizontal distance from d to P. If we translate P by $d_h(d, P)$ unit to the right then the square $abcd$ lies on one side of Edge e.

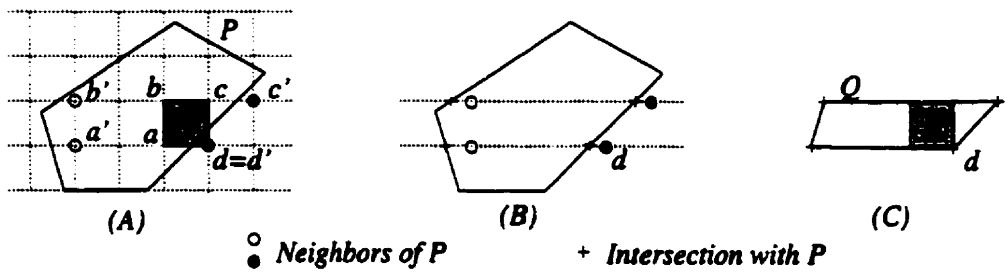

Figure 5.8: Determining if d is a supporting vertex for P. The initial placement of P is shown in (A), while its translated copy passing through d is in (B). Since Q includes $abcd$ (C), d is a supporting vertex.

We want to know if that translation does indeed put the whole square inside P. Suppose that there exists a placement $\tau(P)$ such that $abcd$ lies entirely in P and d is a supporting vertex. Thus there are four intersection points between $\partial\tau(P)$ and

horizontal lines of the grid passing through d and c, as shown in Figure 5.8 (B). One of these points is d. We can compute these points in $O(1)$ time since we have access to the neighbors and their horizontal distances of P. Let Q be the convex polygon passing through these points, as in Figure 5.8 (C). Since $\tau(P)$ is convex, Q lies inside $\tau(P)$. By definition, Q is a trapezoid, where two of its edges lie on horizontal lines of the grid and its left and right boundaries lie on $\tau(P)$. If $abcd$ lies inside Q then d is a supporting vertex. The following is the pseudo-code of our algorithm.

Verify-Case-1-Convex(P, $abcd$)
Input: Convex polygon P and a square $abcd$, where d is its lower right corner.
Output: Determine whether d can be a supporting vertex.
BEGIN
  $\{a', d'\} \leftarrow$ Pair of neighbors, from descending and ascending chains respectively and sharing same y-coordinates as d.
  $\{b', c'\} \leftarrow$ Pair of neighbors, adjacent to a' and d' in their linked lists, resp. and $b'_y = c'_y = d'_y + 1$. See figure 5.8, (A). If there is no such pair, then d has to be treated as c.
  Compute the intersections between ∂P and horizontal lines passing through cd.
  FOR each intersection point
    Add $d_h(d, P)$ to their x-coordinate.
  $Q \leftarrow$ Convex Polygon passing through the updated intersection points.
  IF Q contains $abcd$ THEN
    d is a supporting vertex.
END.

Each step of the algorithm takes $O(1)$ time. The correctness of our algorithm is shown at the end of this section. Determining whether or not c can be a supporting vertex is achieved similarly.

Case 2

We now consider Case 2, where a translation makes a square sharing a point with a descending chain of P. We have to determine if that square is really inside

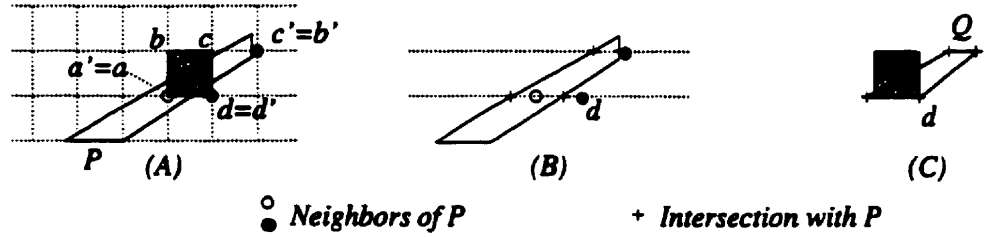

Figure 5.9: Determining if d is a supporting vertex for P. The initial placement of P is shown in (A), while its translated copy passing through d is in (B). $abcd$ does not lie inside Q (C).

P. Again, we have to compare distances from ∂P to the square $abcd$ and also the neighbors computed from an ascending chain. We consider here the case where a is the supporting vertex, which is also the lower left corner of the square. The case of the upper corner is symmetric.

We need to verify if there exists an intersection between $abcd$ and ∂P. This can be done in $O(1)$ time using the neighbors lying on same horizontal lines as the vertices of the square.

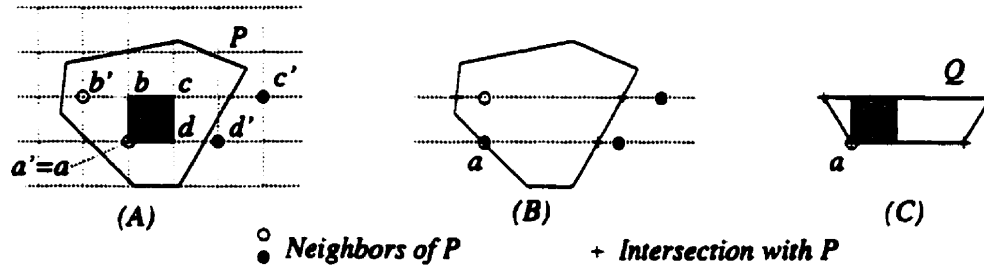

Figure 5.10: Determining if a is a supporting vertex for P. The initial placement of P is shown in (A), while its translated copy passes through a is in (B). $abcd$ does lie inside Q (C).

Verify-Case-2-Convex(P, $abcd$)
Input: Convex polygon P and a square $abcd$.
Output: Determine whether grid point a can be a supporting vertex.
BEGIN
  $\{a', d'\} \leftarrow$ Pair of neighbors, from descending and ascending chains respectively and sharing same y-coordinates as d.
  $\{b', c'\} \leftarrow$ Pair of neighbors, adjacent to a' and d' in their linked lists, resp. and $b'_y = c'_y = d'_y + 1$. See figure 5.10, (A). If there is no such pair, then d has to be treated as c.
  Find the intersections between ∂P and horizontal lines passing through d and c. See Figure 5.10, (B).
  FOR each intersection point
    Add $d_h(q, P)$ to their x-coordinate.
  $Q \leftarrow$ Convex Polygon passing through the updated intersection points.
  IF Q contains $abcd$ THEN
    d is a supporting vertex.
END.

Theorem 5.1 *Algorithm Verify-Case-1-Convex works correctly.*

Proof. Let P be a convex polygon. We have to prove that any neighbor that is not supporting is identified. Let d be that neighbor, being the lower right corner of the square (the proof for the upper corner is similar, by symmetry). If d is the highest neighbor, then Square $abcd$ (as defined previously) cannot be inside P. But the square below $abcd$ can be. Hence, we have to handle the case for the corner c of that square.

It remains to prove that if $abcd$ lies inside P with d on the boundary of the polygon, then there exists a trapezoid Q containing $abcd$. This is easy because we only have to split P into three pieces P_1, P_2 and P_3. The first one lies below the line passing through $\overline{ad}$. Let P_3 be the part of P lying above the line passing through $\overline{bc}$. Let P_2 be the remaining part of P. Let Q be a trapezoid lying in P_2 with its corner being the intersection points between ∂P and lines passing through $\overline{ad}$ and $\overline{bc}$. Q is

a trapezoid because it has two parallel edges and P is convex. P_2 does contain *abcd* since that square is contained by P, but not by P_1, neither P_3. And Q contains *abcd* since that square lies between horizontal lines forming P_2, and P contains *abcd* (this implies that a line segment joining c and d does not intersect its boundary). Since the algorithm computes this trapezoid, in which *abcd* must lie to be in P, it works correctly. ■

Corollary 5.1 *Algorithm Verify-Case-2-Convex works correctly.*

The proof of the correctness of Algorithm Verify-Case-2-Convex is similar to the previous one. Both algorithms are using a trapezoid to determine whether or not a given square lies inside P. The coordinates of that trapezoid are computed in $O(1)$ time since we have an easy access to the set of nearest neighbors. The running time of our algorithms is proportional to the number of neighbors.

Corollary 5.2 *We can determine neighbors that cannot support a square inside a convex polygon in $O(h)$ time, where h is the height of the polygon.*

5.1.5.2 Simple Polygons

In this section we explain how to identify nearest neighbors that can support a grid square for simple polygons. The difference with convex polygons is the number of neighbors on the same horizontal line may be greater. Figure 5.11 shows an example where we cannot apply directly the trick of the trapezoid explained in the previous section. We show how to determine if a square lies inside a simple polygon in $O(\log m)$ time with a $O(m \log m)$ time preprocessing step.

Let P be a simple polygon with m vertices. Again, we consider only the part of the polygon that lies between lines passing through two consecutive horizontal lines of the grid. As shown in the figure, some vertices of P may lie inside the square. For a given square, we have to verify if some vertices lie inside it. For this purpose we build a two-dimensional array. The number of rows is $\lfloor h \rfloor$. The number of columns is $m-2$ in the worst case. Each row contains vertices of P lying between two consecutive horizontal lines of the grid. Determining which vertices belong to which row takes

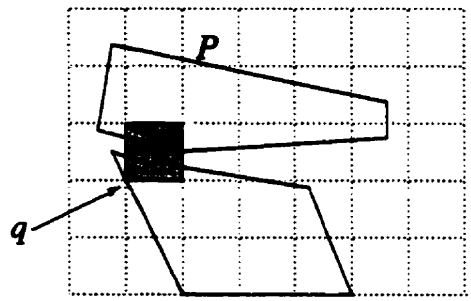

Figure 5.11: Grid point q is not a supporting vertex for the polygon.

$O(m)$ time. Each entry in the array has at most one vertex of P. Each row is sorted by x-coordinate. Therefore, it takes $O(m \log m)$ time to build this two-dimensional array. The following is the pseudo-code of our algorithm.

Verify-Cases-1-2-Simple(P, *abcd*, q, V)
Inputs: Simple polygon P, a square *abcd*, and two-dimensional array V containing the vertices of P.
Output: Determine whether grid point q (a vertex of the square) can be a supporting vertex.

```
BEGIN
    n ← Number of vertices of P lying between lines passing through ad and bc.
    IF n = 0 THEN
        Use a O(1) time algorithm from the previous section (convex case).
        This verifies if there is an intersection between ∂P and abcd.
    ELSE
        FOR each point lying between ad and bc
            Add d_h(q, P) to its x-coordinate.
        IF abcd contains a point OR intersects an edge THEN
            q is not a supporting vertex.
        Use a O(1) time algorithm from the previous section (convex case) to verify
        if there is an intersection between ∂P and abcd.
END.
```

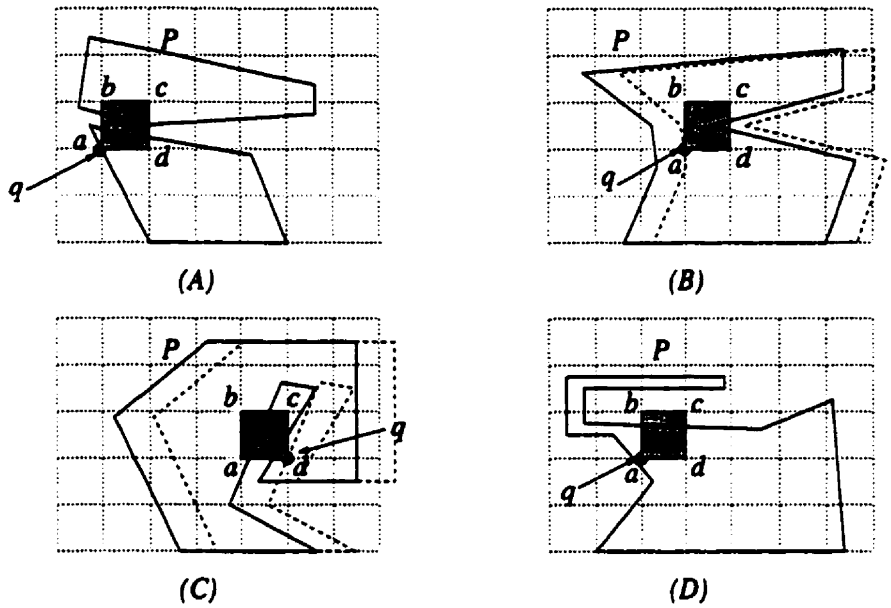

Figure 5.12: Some instances where Verify-Cases-1-2-Simple has to deal with.

Figure 5.12 shows some instances of the problem of finding which neighbor cannot be a supporting vertex. In (A), q is not such a vertex since ∂P intersects the square. This case is handled in the ELSE statement of the algorithm, which finds that there is one vertex of P inside the square. In (B), q can be a supporting vertex because a translation to the right of P makes *abcd* empty. This case is handled in the ELSE statement of Algorithm Verify-Cases-1-2-Simple and then by an algorithm for the convex case. We also need this algorithm because there may be intersection between *abcd* and ∂P. In (C), this algorithm finds that q can be a supporting vertex (from an ascending chain) using the neighbors. A translated copy of P is shown as a dashed polygon in the figure. The last example (D) is handled by the ELSE statement and the algorithm finds out that an edge (having endpoints lying between l_1 and l_2) intersects *abcd*. The case where there is an edge with only one endpoint between l_1 and l_2 is handled by algorithms from the previous section. These methods compute intersection between that kind of edge using its neighbor. Such a neighbor exists because the edge intersects a horizontal line. This leads to the following result

Theorem 5.2 *Algorithm Verify-Cases-1-2-Simple works correctly.*

Proof. Let P be a simple polygon and q be a neighbor of P. Let l_1 and l_2 be two consecutive horizontal lines on the grid where q passes through one of those. Let $abcd$ be a square lying between these lines. First, we consider the case where there is no vertex of P lying between l_1 and l_2. If $abcd$ lies inside P, algorithms from the previous section find the correct answer because there are only two edges of P crossing the region between l_1 and l_2. Similarly, if $abcd$ intersects ∂P then algorithms for the convex cases work correctly.

Now, we consider the case where there is at least one vertex of P lying between l_1 and l_2. If there is a vertex of P lying inside $abcd$ then this case is handled by the algorithm. Otherwise, we also need to verify if there is an edge of P with its endpoints lying between l_1 and l_2 intersecting $abcd$. If so, the algorithm returns true and then q is not a supporting vertex. It remains to verify if there is another edge of P intersecting the square. This part is handled correctly by the algorithm for the convex case. In fact, if there is an intersection then it is found by these algorithms using the nearest neighbors of P. Since all cases are successfully handled by the algorithm then it works correctly. ∎

Corollary 5.3 *We can remove nearest neighbors that cannot support a square inside a simple polygon in* $O((m + hl)\log m)$ *time.*

We have shown how to determine neighbors that cannot support a square of P. Since each critical event involves one grid square, we have to make sure that there is at most two events for a given square. If P has a vertical edge, we need to know the number of squares that may intersect ∂P. This may be different from the number of nearest neighbors for that edge. Therefore, for each square that "enters" P, there is exactly one event describing it. This is similar for a grid square that "leaves" P during its motion. This can be determine in $O(hl)$ time.

5.1.6 Finding an Optimal Translation

Before finding an optimal placement of the polygon, we have to sort the set of critical events by their horizontal distance to P, in increasing order. Sorting the set of nearest neighbors is achieved by the algorithm presented in Chapter 4. Once the sorting step is done, we can successively apply each translation and count the number of squares inside P. In fact, we don't need to count squares at each translation because we already know how many squares we have in the initial placement. We can use the type of each event: whether it comes from an ascending chain (AC, case 1) or a descending chain (DC, case 2). Thus, updating the number of squares inside P can be done in $O(1)$ time at each translation. We explain in this section the final step of Algorithm A1.

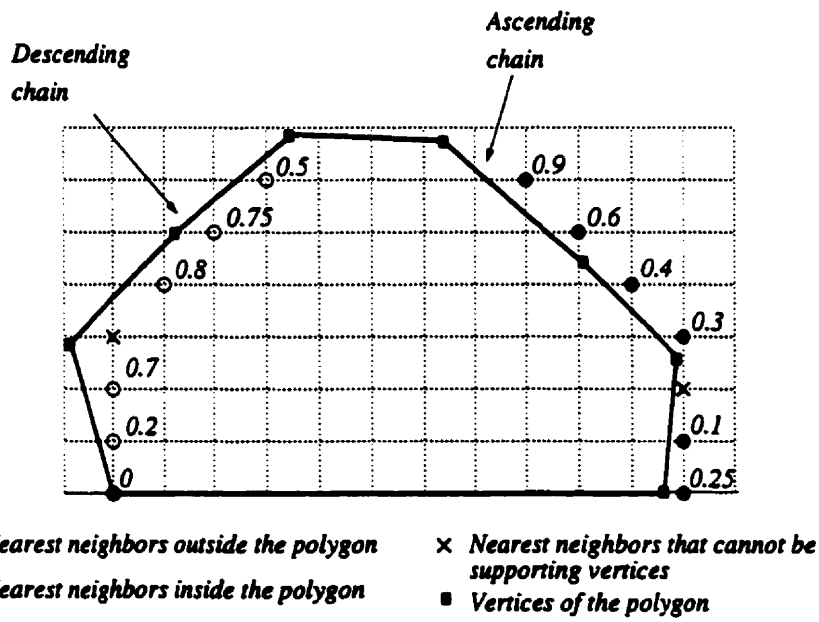

Figure 5.13: An initial placement of a convex polygon on a grid. It contains 48 intact squares.

Refer to Table 5.1 and Figure 5.13. The table gives distances from a convex polygon P to its nearest neighbors that can be supporting vertices, with their types as shown in the figure. Note that all distances are different. The information given in

tables 5.1 and 5.2 are the following. The first row shows the index of an event. The second row refers to the horizontal distance between an event i and P. The next row displays the distances by which the polygon has been translated to the right, at each step. The fourth row states the type of ith event and the last row of the table shows the number of intact squares when P is translated such that to represent the current event. The initial placement of P contains 48 intact squares.

i	1	2	3	4	5	6	7	8	9	10	11	12
Init. dist.	0.0	0.1	0.2	0.25	0.3	0.4	0.5	0.6	0.7	0.75	0.8	0.9
Transl.	0.0	0.1	0.1	0.05	0.05	0.1	0.1	0.1	0.1	0.05	0.05	0.1
Type	DC	AC	DC	AC	AC	AC	DC	AC	DC	DC	DC	AC
Squares	48	48	48	48	49	50	50	50	50	49	48	48

Table 5.1: Scanning the set of critical events having different distances, from Figure 5.13.

One might think that AC corresponds to the set of translations which put a new square inside P. Applying a first translation of 0.1 units does not increase the number of squares, because we also lost one. Note that other distances decrease by 0.1 units (third row) because the whole polygon is moving to the right (towards the set of nearest neighbors). The third translation puts square sharing a vertex with ∂P. The fourth translation brings a new square inside P but we know that we lost the square from translation #3. Thus we still have 48 intact squares in P. Translations #5 and #6 put two new squares inside P. The total is then 50. We did not lose any square because we did not visit an event from a descending chain. Figure 5.14 shows an optimal placement.

Since we know how many squares lie in P at the beginning of the process then we only have to scan the set of sorted distances in linear time. By looking at the type of the current event, we can increase or decrease if necessary the number of squares inside P. This is true because an event of type AC will bring a new square inside the polygon. Thus we can apply only one translation corresponding to the maximum. In the previous example shown in Table 5.1, translating the polygon to the right by 0.4

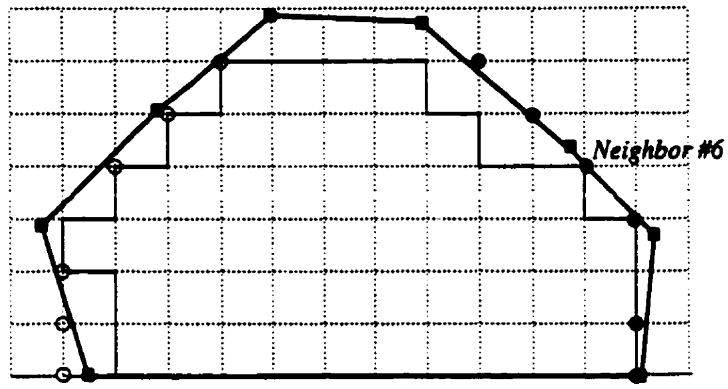

Figure 5.14: An optimal placement of the instance shown in Figure 5.13. The polygon has been translated by 0.4 units and now contains 50 squares.

units from its initial placement gives an optimal solution. This is due to the fact that all critical events have been found and verified using the set of nearest neighbors, and reflex vertices of P, if any. Section 5.1.7 shows the correctness of the algorithm.

Now we take a look at an example where different critical events lie at same distances from the polygon. It is shown in Table 5.2. Even if the events are sorted by distances, they are not sorted by their types when some of those have same distances. At a given distance t, we may have to deal with more than 1 event. Let the current placement be $t = 0.5$. We will lose one square if we translate to the right because the current step tells us that we have an event from DC. Thus, translating P by 0.1 units brings three squares sharing at least one vertex with ∂P. Let the current placement be $t = 0.6$. We lost one square and have gained a new one. We can determine in $O(k)$ time how many squares we will lose if we translate to the right, where k is the number of events lying at same distances from P. We can also determine the number of squares lying inside P by looking at the number of squares lost and to the event of type AC.

An optimal solution from table 5.2 is given by neighbor #3, located at 0.3 units from the polygon. The pseudo-code of the scanning process is shown on the next page.

i	1	2	3	4	5	6	7	8	9	10	11	12
Initial dist.	0.1	0.2	0.3	0.3	0.4	0.5	0.6	0.6	0.6	0.7	0.8	0.8
Translation	0.1	0.1	0.1	0	0.1	0.1	0.1	0	0	0.1	0.1	0
Type	AC	AC	AC	DC	DC	AC	AC	DC	DC	AC	AC	DC
# of squares	35	36	37	37	37	37	37	37	37	35	36	36

Table 5.2: Scanning the set of critical events with similar distances.

Scanning-Critical-Events(E, n)

Input: A sorted array of critical events with their types and distances, and the number of squares lying inside the polygon. E contains only critical events.

Output: Solution to Problem 1.

BEGIN

Initialize *current* and *best* to n and *current* respectively.

BestTranslation $\leftarrow$ 0. *CurrDistance* $\leftarrow$ Distance($E[1]$).

lost $\leftarrow$ For the current distance, find how many squares will be lost in the next translation.

FOR i from 2 to the size of E

current $\leftarrow$ *current* $-$ *lost*.

$\{new, lost\}$ $\leftarrow$ For the current distance, find how many new squares have been added inside P and will be lost in the next translation.

current $\leftarrow$ *current* $+$ *new*.

Update *best*, *CurrDistance* and *BestTranslation* if necessary.

Return *best* and *BestTranslation*.

END.

Where the function Distance returns the horizontal distance of Event $E[i]$ from P. The variables *current*, *new* and *lost* are integers and contain the current number of squares inside P, added inside P and lost respectively. *CurrDistance* keeps track of the current distance. *Best* and *BestTranslation* contains the maximum number of squares and the optimal translation.

The next theorem states that the previous algorithm finds an optimal solution to Problem 1, given the set of critical events.

Theorem 5.3 *Scanning-Critical-Events finds an optimal solution to Problem 1.*

Proof. Let P be a simple polygon. The proof is by induction on the number of critical events. If P does not have such an event then the current placement is optimal. If there is only one critical event q, we translate the polygon such that q is met. If q comes from an ascending chain then a new square has been added inside P. The current number of squares inside P is increased so it is optimal. If q is of type DC, then the number of squares inside P does not change.

Suppose that the algorithm works correctly when the number of events is n. We have to prove that the algorithm works correctly when the number of events is $k = n+1$. Let q be the kth event lying at distance d from P. Let $c \geq 1$ be the total number of events lying at distance d from P. By the induction hypothesis, the algorithm finds an optimal solution for the first $k-c$ events. Also, the algorithm computes correctly the number of squares that will be lost in a next translation.

Translate P such that q is encountered on its boundary. First the algorithm subtracts the number of squares that get lost. The next operation is counting the number of new squares added inside P. The type of Event q is verified during that step, as well as the other neighbors lying at distance d from the original placement of P. By the induction hypothesis, the algorithm works correctly and find an optimal placement to Problem 1. ∎

5.1.7 Worst Case Analysis

Let P be a convex polygon with m vertices. The supporting wall W is given and is horizontal. The highest vertex of P lies at h units from W.

Theorem 5.4 *Algorithm A1 solves Problem 1 and runs in $O(m + h \lg m)$ time where h is the height of the convex polygon containing m edges with respect to the given supporting wall W.*

Proof. The following are the main steps of the algorithm and their running time. The first step involves rotating the polygon P such that one of its edges, W, is a horizontal supporting wall. The leftpoint of W must lie on a grid point. This takes $O(m)$ time because P has m edges. Computing the number of intact squares inside P can be done in $O(h)$ time, as shown in Section 3.2. It takes also $O(h)$ time to compute the set of nearest neighbors of P (Section 5.1.2). Finding the set of nearest neighbors that cannot be supporting vertices takes $O(h)$ time. An important part of the algorithm is the sorting step, which takes $O(h \lg m)$ time because we have to merge m sorted lists. Those lists are sorted in $O(h)$ each. Scanning the sorted list of neighbors takes $O(h)$ time. Therefore, it takes $O(m+h \log m)$ time to find an optimal solution when the polygon is convex and the supporting wall is given. ∎

Let P be a simple polygon with m vertices where r of those are reflex vertices. The supporting wall W is given and is horizontal and the highest vertex of P lies at most $\lceil h \rceil$ units from W.

Theorem 5.5 *Algorithm A1 solves Problem 1 for simple (nonconvex) polygons and runs in $O((m+hl)\log m)$ time.*

Proof. Placing the polygon P on the grid such that W is a horizontal supporting wall and its leftpoint lies on a grid point takes $O(m)$ time. We have to count the number of intact squares inside P and compute the set of nearest neighbors. This takes $O(hl)$ time because P has l local minima and height h. Finding the set of nearest neighbors that cannot be supporting vertices takes $O((m+hl)\log m)$ time since we have to spend $O(\log m)$ time for each of them. Note that a preprocessing step of $O(m \log m)$ is necessary. Sorting the remaining neighbors with respect to their horizontal distances with the edges of P (increasing order) takes $O(hl \log m)$, as shown in Chapter 4. Computing the set of vertices of P introducing critical events can be done in $O(r \log r)$ time. Finding an optimal solution to Problem 1 takes $O(hl)$ time because there are at most $O(hl)$ critical events to verify. Therefore, it takes $O((m+hl)\log m)$ time to find an optimal solution when the supporting wall is given. ∎

Observe that if a simple polygon does not have any local minimum, the complexity of Algorithm A1 is $O((m+h)\log m)$.

5.2 Allowing y Translations (Algorithm A2)

We now consider the problem when both x and y translations of the polygon are allowed. For the convex case, we present an algorithm running in $O((m+h^2+w^2)\log(m+h+w))$ time, where m, h and w are the number of vertices, height and width of the polygon. If the polygon has r reflex vertices our algorithm runs in $O(m(h+w)+(m+r^2(h^2+w^2))\log(m+rh+rw))$ time.

5.2.1 Definition of the Problem (revisited)

Problem 2 (2 degrees of freedom) *Let P be a simple polygon and $\mathcal{G}$ a grid. Find a translation of P on $\mathcal{G}$ such that the number of intact squares from the grid inside P is maximum.*

The difference with the problem solved in Section 5.1 is that y-translations are also allowed. Using xy-translations to find an optimal placement of P on the grid $\mathcal{G}$ increases the number of nearest neighbors. We first define an initial placement of P where P has an edge which lies on a horizontal line of the grid and its left endpoint is the origin of P. Figure 5.15 shows the region R of the grid where the set of nearest neighbors of P lie. These are grid points that may be hit by ∂P for a given translation.

We now explain how to compute this region of the grid. The origin (i.e. a chosen vertex) of four translated copies of P are placed on four grid points belonging to the same grid square. Let P_0, P_1, P_2 and P_3 be those copies. The smallest axis-parallel bounding box containing these polygons is computed in order to simplify the computation of Polygon U, the union of all possible translated copies of P, where a translation of (x,y) units is at most $(1,1)$. The intersection of the four copies of P is labeled I. R is the region where we find some nearest neighbors of P, that can be hit by the polygon while translating. As seen in Section 5.1, once a nearest neighbor

hits ∂P, one of its vertices hits a grid square and the number of intact squares inside P is subject to change.

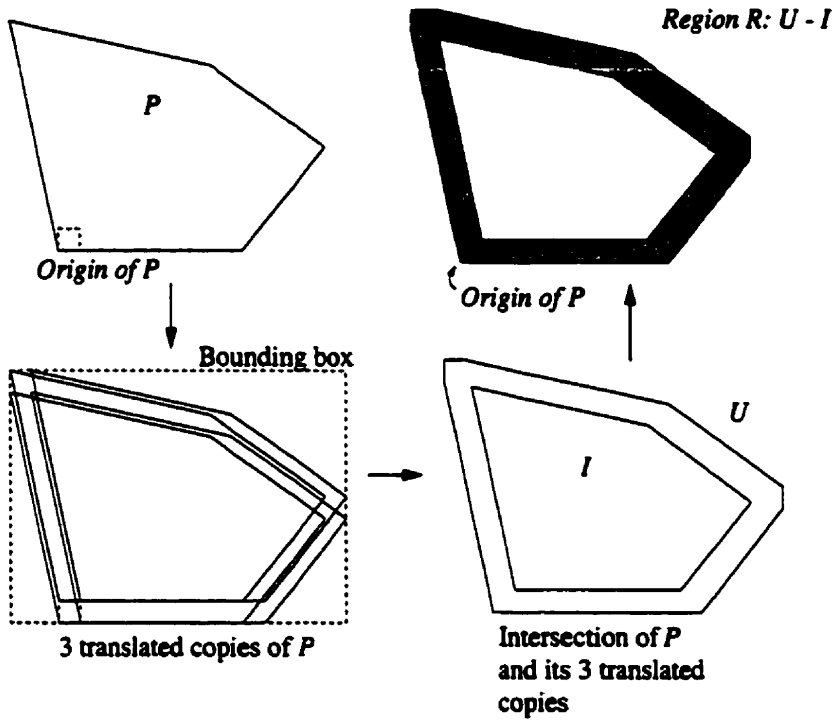

Figure 5.15: Preprocessing steps of the computation of the optimal solution for Problem 2. Region R contains the nearest neighbors of P.

5.2.2 Main idea of A2

We limit translations to at most 1 unit in x and 1 unit in y, because any translation (u, v) where $u \geq 1$ and $v \geq 1$ can be expressed as a translation of $(x = u - \lfloor u \rfloor, y = v - \lfloor v \rfloor)$. This is due to the periodicity of the grid. Let O be the origin of P and let q be a neighbor of P. A neighbor of P is a grid point that may be hit by its boundary during a translation. Let s_O be the grid square where O lies on its lower left corner, in the initial placement of P. Note that a translation of P by (u, v) keeps the origin O inside or on the boundary of s_O. Also, translating P along a neighbor makes O move

along a polygonal chain C, which lies in s_O. C splits s_O into two simple polygons R_1 and R_2. If we place O on C, then q lies on ∂P, by the definition of C. However, if we place O in R_1 (R_2) then P contains (does not contain) q. This is shown in Figure 5.16.

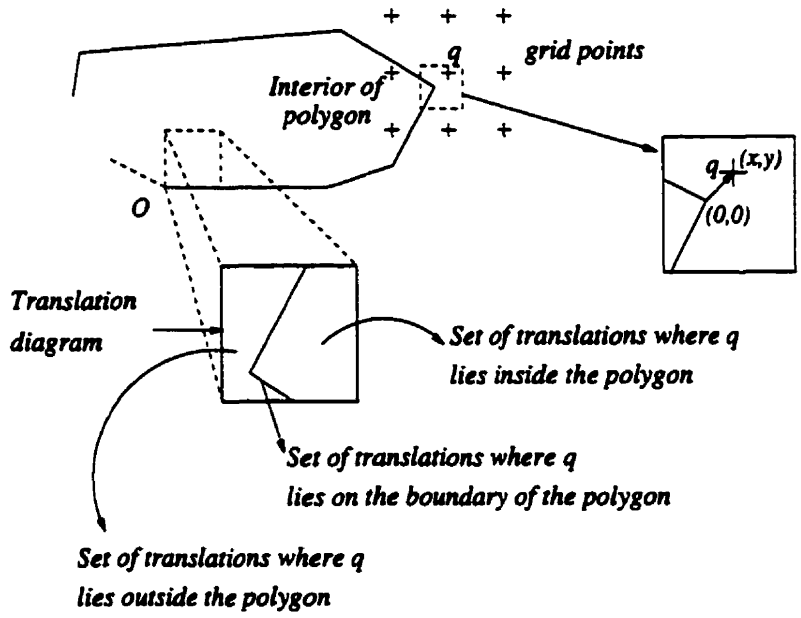

Figure 5.16: Example of a polygon and the translation diagram, representing the possible placements of the polygon on the grid such that q lies on its boundary, inside it, or outside it.

However, when a grid point q lies on the boundary of the polygon this does not mean that it is a supporting vertex. But if q is a supporting vertex for the square s_q, then there exists a set of placements of P such that s_q lies inside P. We define this set of placements as a *containing region*. This region is a simple polygon representing the set of translations of P such that s_q lies entirely inside it. Figure 5.17 shows the containing regions for a set of squares. These squares may lie entirely inside P for some placements of the polygon but may intersect P for other placements. As you can see, there is also a set of squares from the grid (shown grey in the figure) that lie entirely inside P for any translation by at most one unit of the polygon.

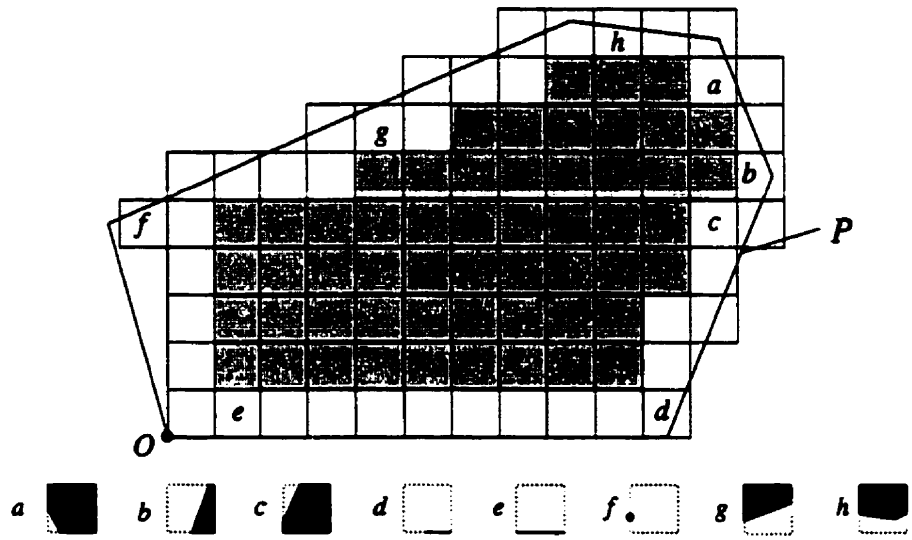

Figure 5.17: An initial solution to Problem 2. Possible translations are shown with darker polygons for Squares a, b, c, d, e, f, g and h. Placing one of these (say h) on the square having Corner O, gives all the possible placements of P such that the corresponding grid square h lies inside P.

The remaining squares lie near the boundary of the polygon. Observe that for a square lying inside P for any translation, the corresponding containing region is a unit square.

Computing these containing regions allows us to apply a sweep line algorithm to find an optimal solution to Problem 2. We define the translation diagram as the set of containing regions of all squares that may intersect ∂P, for some translation. Finding a point (x, y) in the diagram of maximum depth gives us an optimal solution to Problem 2. In other words, such a point lies in a maximum number of containing regions. The advantage of using the containing regions is that we do not have to remove a set of nearest neighbors that cannot be supporting vertices, as we did for the first problem in Section 5.1.

Algorithm A2 works in the following way. The solutions visited have at least two contacts between the polygon and grid squares, since an optimal placement of P has two contacts, as explained in Theorem 3.3, Section 3.4. This allows us to visit only

a finite number of solutions, represented by critical events. We keep these events in a queue. The translation diagram is the arrangement of all containing regions. An example is shown in Figure 5.26. More details on the way it works and its complexity are given in the next sections.

Algorithm A2(P)
Input: A simple polygon P having an edge parallel to the x-axis.
Output: A placement of P on the grid such that the number of intact squares inside P is maximized, given P can be translated in x and y.
BEGIN
  $e_k \leftarrow$ horizontal edge.
  Let the origin O of P be the smaller x-coordinate of e_k.
  Translate P such that O lies on a grid point.
    This corresponds to an initial placement of P.
  Find the number of squares that stay inside P for any small translations.
  $\mathcal{D} \leftarrow$ Compute the translation diagram.
  Initialize the event queue.
  Initialize the sweep line status as a balanced binary search tree.
  Apply a Line Sweep algorithm on $\mathcal{D}$ to find an optimal placement.
    Each visited solution is of the form explained in Section 3.4.
  Return an optimal placement and the number of squares.
END.

The next section shows how many squares may be intersected by the boundary of the polygon. Sections 5.2.4 and 5.2.5 show how to compute the translation diagram for convex and nonconvex polygons respectively. Section 5.2.6 explains how to find an optimal placement using the translation diagram and a sweep line algorithm. The number of intersection points in the diagram is discussed in Section 5.2.7. The last section is about the correctness of Algorithm A2 and its complexity.

5.2.3 Number of Nearest Squares

For each grid point lying in Region R (see Figure 5.15), there exists at least one line segment (the degenerate case is a point) in the translation diagram. First, we want to know how many squares from $\mathcal{G}$ can "enter" inside P or may "exit" P during its translation. If P is convex then the number of squares to consider is $O(h+w)$ where h is the height of the polygon, and w is its width. Observe that the translation of P to the right (by at most 1 unit) hits at most $\lceil 2h \rceil$ squares. Also, a y-translation by one unit makes the polygon hits at most $\lceil 2w \rceil$ squares.

If P is nonconvex then the number of squares to consider is $O\left(r(h+w)\right)$, where r is the number of reflex vertices in P. In fact, a monotone chain may hit $\Omega(h+w)$ squares. This implies that the upper bound $O\left(r(h+w)\right)$ is tight.

Those squares can be easily identified by modifying the algorithms that count the number of intact squares from Sections 3.2 and 3.3 for convex and nonconvex polygons respectively. We can also find the number of squares that lie inside P for any translations (grey squares in Figure 5.17) with these algorithms.

5.2.4 Computing the Translation Diagram for Convex Polygons

We now show how to compute the translation diagram $\mathcal{D}$ for convex polygons. In fact, this step involves only computing the set of containing regions. This takes $O(m+h+w)$ time since we have to spend linear time for each edge of P, with respect to the number of grid points being hit. The number of regions to compute is $O(h+w)$. The arrangement of all regions contains $O(h^2+w^2)$ polygons. This is explained in Section 5.2.7. Each grid point produces at least one line segment in $\mathcal{D}$. The total number of line segments involving vertices of P is m. In other words, if there are n nearest neighbors producing exactly one line segment then the number of segments in the diagram is $O(m+n)$. We explain how to find the containing region of one square, since the diagram is an arrangement of regions. For clarity, we consider the following cases for the squares that may intersect ∂P, with respect to an initial placement of P. Note that these cases may overlap.

1. Square already lying entirely inside P.
2. Square intersecting ∂P.
3. Vertices of P involved in the computation of a containing region.

Case 1: Square lies entirely inside P.

Refer to Figure 5.18. Let s be a square of the grid lying inside P. We want to compute the containing region for that square. Let e be an edge of P that intersects s for some translation. Let O_e be the lower endpoint of e and s' be a unit square whose bottom left corner is O_e (s' is not necessarily a grid square). Let's do an x-translation (or y-translation) of e such that it hits s, and s still lies on one side of e. Let l be a line passing through e. l splits s' into two convex regions. One of this region represents the containing region of s, i.e. when e does not intersect s. Computing this region can be done in $O(1)$ time, since we deal with $O(1)$ size objects and we can use nearest neighbors, which give us fast access to the edge involved. If s is intersected by only one edge then we are done. Otherwise, we refer the reader to the third case.

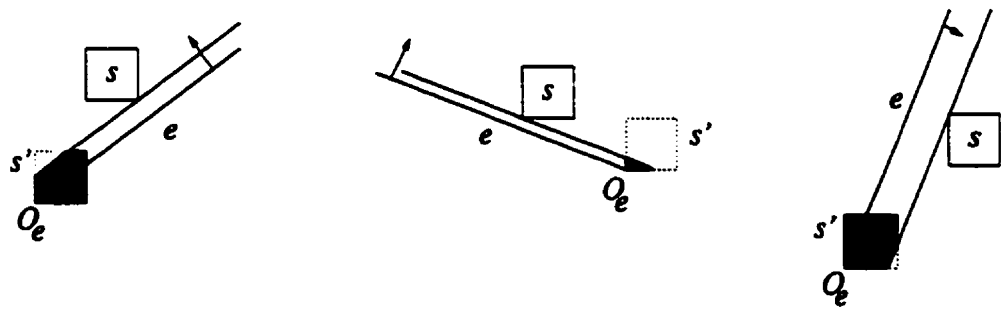

Figure 5.18: Three examples of computing the containing region of a square s. One edge e is involved and s does not intersect e. The arrows show on which side of e the interior of the polygon is.

Case 2: Square intersecting ∂P.

The method is similar to Case 1, explained above. Examples are shown in Figure 5.19. Let s be a square of the grid intersecting an edge e of P. Let O_e be the endpoint of e with the smaller y-coordinate and s' be a square whose bottom left corner is O_e.

The first step is to find a x-translation (or y-translation) of e such that it hits a corner of s where s lies on one side of e. Let l be a line passing through e. l splits s' into two convex regions. One of this region represents the containing region of s, i.e. when e does not intersect s. This takes $O(1)$ time to compute. If s is intersected by only one edge of P then we are done. Otherwise that square belongs to Case 3.

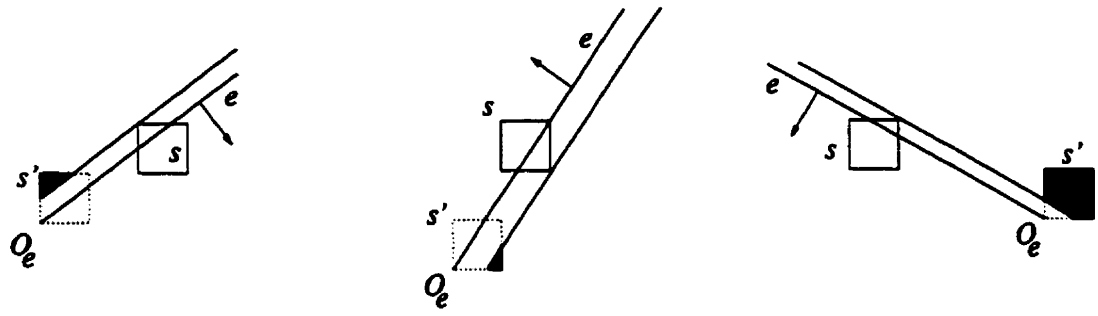

Figure 5.19: Examples of computing the containing region of a square s. One edge e is involved and intersects the interior of s. The arrows show on which side of e the interior of the polygon is.

Case 3: A vertex of P is involved in the computation of the containing region of a square s.

Suppose that there is only one vertex of P involved, which means that Square s is intersecting two edges e_1 and e_2 for some translations of P. We split the computation of the containing region into two parts. We first compute the containing region for s with e_1 and then with e_2, as shown in Figure 5.20. Then the intersection between these two regions is computed which gives the containing region for s. However, if more than two edges of P are involved, we only have to repeat the process with those edges. Determining how many edges of P are necessary to compute the containing region for a square takes $O(1)$ time, if the set of nearest neighbors are given as explained in Section 5.1.2. It is possible that the edges of P may not be long enough to intersect s. If it is the case, then we only have to use lines passing through these short edges.

The correctness of our methods is shown in the next lemma:

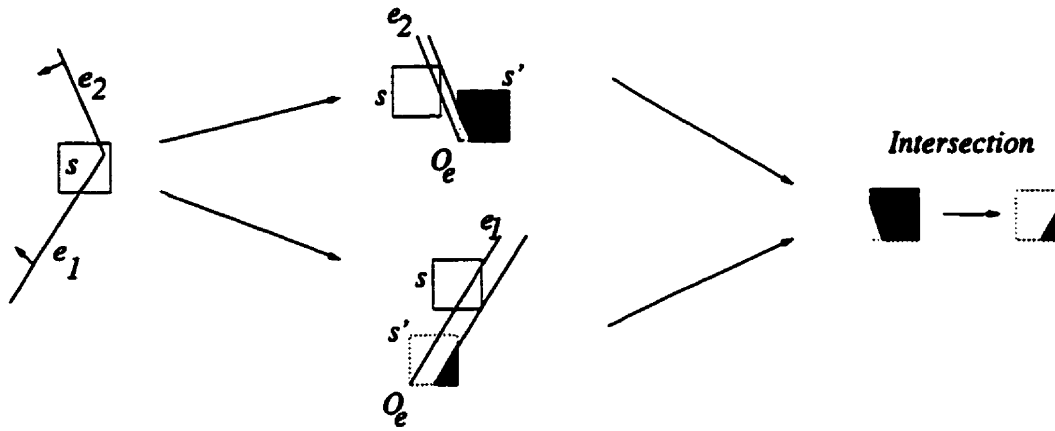

Figure 5.20: Example of computing the containing region of a square s, involving two edges e_1 and e_2 of a convex polygon.

Lemma 5.1 *The containing region of a square s from $\mathcal{G}$ is given by the intersection of the containing region of its corresponding edges.*

Proof. (By induction on the number of edges involved)
If there is only one edge intersecting Square s for any translation of P, then the containing region for s is determined by the containing region for that edge, as explained in Cases 1 and 2. No other edge of P intersects s. Suppose that the statement of the lemma is true when the number of edges involved is n, we prove that the containing region for Square s having $n+1$ corresponding edges is given by the intersection of the containing regions of these edges.

Let e be the $(n+1)th$ edge. If we remove that edge, then the containing region of s is correctly given by the intersection of the containing regions of the other n edges. Let A be that containing region of s. We then compute the containing region B of s with Edge e, given by Case 1 or 2. We know that for any placement of P given by B ensures us that e does not intersect s. This is also true for A. Let $C = A \cap B$. By definition of A, B and C, a point lying in C corresponds to a placement of P on $\mathcal{G}$ such that the edges involved in A and B do not intersect the interior of s. This completes the proof. ∎

Lemma 5.2 *The containing regions of a convex polygon P can be computed in $O(m+h+w)$ time.*

Proof. There are $O(h+w)$ squares to consider. These can be found in $O(h+w)$ time using algorithms from Section 3.2. For a square s, if there is only one edge of P that may be intersected, then its containing region can be computed in $O(1)$ time, as explained in Cases 1 and 2. Let $n \leq 4m$ be the number of squares that may intersect more than one edge of P. We can identify these squares in $O(m)$ time, using the vertices of P. Since each of these squares intersects more than one line segments and the total number of edges in P is m, we compute the containing regions of these squares in $O(n+m)$ time. ∎

5.2.5 Computing the Translation Diagram for Nonconvex Polygons

If P is nonconvex, then we have to consider $O(r(h+w))$ grid squares. Refer to Figure 5.21. Two nonconvex polygons are shown. Let s be a grid square lying inside P. Computing the containing region of s involves $O(m)$ edges for both polygons, where m is the number of edges in them.

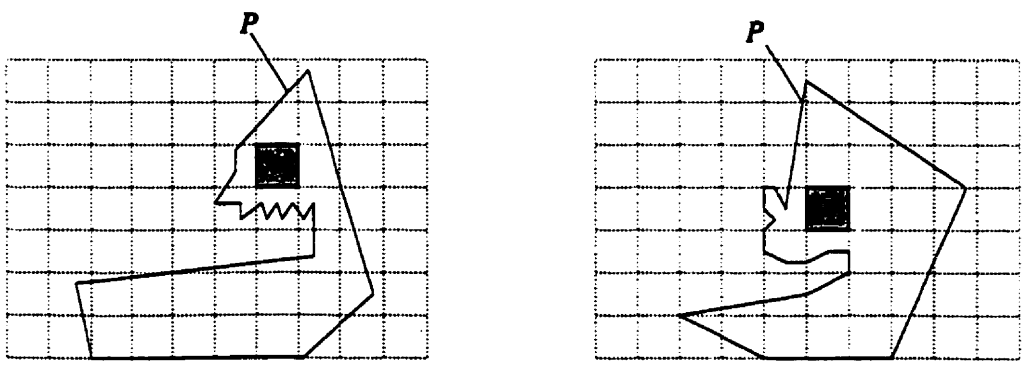

Figure 5.21: Two simple polygons with 18 edges. Computing the containing region of Square s involves 11 edges (12 edges on the right).

A vertex of P is involved in at most four computations of containing regions. For each pair of edges sharing an endpoint we can compute in $O(1)$ time the containing region of a given square. Hence we need a total of $O(m)$ time for this. Observe that we can easily determine, for a square, which vertices are involved. This can be achieved using the set of nearest neighbors. However, a square can be intersected by $O(r)$ edges of P, as shown in Figure 5.22. Also, each edge of P can intersect at most $O(h+w)$ grid squares. An edge intersecting a square adds $O(1)$ vertices to the containing region of that square.

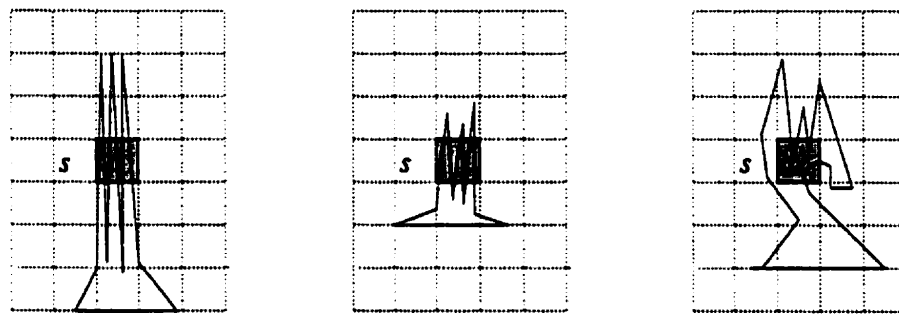

Figure 5.22: Polygons where the number of edges and vertices that may intersect Square s for any small translations is $O(r)$.

If a square is intersected by $O(m)$ vertices of P, then only $O(1)$ squares have a containing region with $O(m)$ vertices. Let s be a square for which the vertices of P are not involved in the computation of its containing region. At most four edges of P have an impact on the containing region of s, even if $O(m)$ edges can intersect s. This is due to the fact that vertices of P have not been involved, and we translate P in two directions only. Other edges are either below, above, on the left or right of the edges having an impact. Hence, the containing region of Square s has $O(1)$ vertices. Figure 5.23 shows an example, edges involved in the computation of the containing region are identified with arrows.

For each edge, we can compute the corresponding containing region for the squares it intersects. Once edges have been processed, we can compute in $O(m)$ time, for each square, the intersection of all regions. However, only $O(1)$ squares have containing

regions with that many vertices.

Since we have m edges, and each of them intersect $O(h+w)$, we compute the containing regions in $O(m(h+w))$. Correctness follows from Lemma 5.1.

Lemma 5.3 *The containing regions of a nonconvex polygon P can be computed in $O(m(h+w))$ time.*

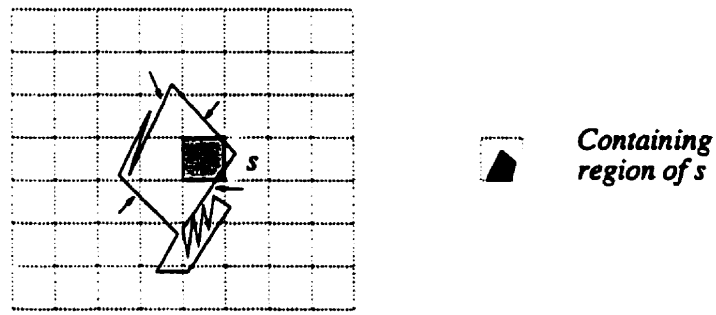

Figure 5.23: Polygons where $O(m)$ edges can intersect Square s. There is no vertex of P involved in the computation of the containing region. Four edges have an impact on the containing region of s.

5.2.6 Finding an Optimal Solution

In this section we show how to use a sweep line algorithm to find an optimal placement to Problem 2. First, we discuss a property of the translation diagram $\mathcal{D}$. Let c_i be a cell of $\mathcal{D}$. The points lying in c_i correspond to placements of P on the grid where the number of intact squares inside P does not change. This is shown in the next lemma.

Lemma 5.4 *Let the origin O of P be in cell c_i of $\mathcal{D}$. Translating the origin of P to any point in c_i does not change the total number of squares inside P.*

Proof. Suppose that the total number n of squares changes while we are moving the origin O of P inside Region c_i of the diagram. First assume that we are losing a square s. This means that at least one vertex of s lies outside Cell c_i. That vertex

was inside P or on the boundary of P before the translation. Hence a line segment should have been crossed. This yield a contradiction. The argument is the same if n increases. This proves that the number of squares inside the polygon does not change if O still lies inside c_i before and after a translation. ■

We only have to traverse all regions of the translation diagram in order to find an optimal solution. The main idea is to use a sweep line algorithm [O'R98, CLR90]. A vertical line L sweeps $\mathcal{D}$ from left to right, and at each critical event, updates the best solution visited so far. Two line segments intersecting correspond to a critical event. Intersections between line segments are computed during the sweep. Theorem 3.3 allows us to visit solutions with at least two contacts. Hence, our sweep line algorithm visits intersection points in the translation diagram. Such points in $\mathcal{D}$ involve two edges of P with their supporting vertices, verifying Theorem 3.3.

Our method is similar to the sweep line algorithm from Dickerson and Scharstein [DS98]. Their algorithm scans a set of containing regions in order to determine a point of maximum depth (a point lying in a maximum number of regions). It uses a segment tree to store the boundaries of the containing regions. In their case, the boundaries are sine curves. They have modified the segment tree to handle this kind of boundaries. Since we deal with translations only, the boundaries of the containing regions are polygonal lines.

5.2.6.1 Event Queue

In this section we discuss the complexity of the event queue, used by our line sweep algorithm.

Convex Polygons

Because L is moving only to the right, we have to sort the endpoints of the set of line segments. Thus, the queue is represented as a heap. This allows us to update this structure in logarithmic time in the number of segments in the diagram.

We know that for any x-coordinate of L, the number of cells intersecting L is at most $O(m+h+w)$ since P is convex, as explained in Section 5.2.4. Thus the size of the event queue is at most $O(m+h+w)$ since endpoints of edges from the containing

regions are appearing in it. Everytime that we test if two boundaries intersect, at most one critical event is inserted into the queue (and one event is removed). Also the number of intersection points when L is at the origin of the diagram is $O(m+h+w)$. Section 5.2.7 shows an upper bound on the number of intersection points to consider.

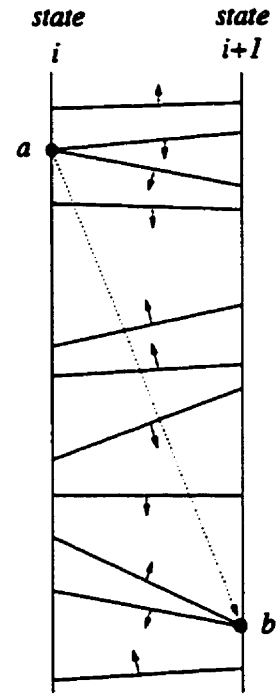

Figure 5.24: Two consecutive states of the line sweep. Arrows show on which side of the line segment is the interior of the containing region. 2 squares have been lost and 4 have been added during the translation of P, from a to b.

Nonconvex Polygons

Initializing the queue involves at most $O(m + r(h + w))$ line segments, because the sweep line L intersects at most $O(m+r(h+w))$ boundaries of containing regions. Also, the number of squares to deal with is $O(r(h + w))$. At any time during the sweeping of L to the right, there are at most $O(m + r(h + w))$ elements in the event queue. The next section discusses the sweep line status.

5.2.6.2 Sweep Line Status

A sweep line algorithm also needs a data structure to maintain the intersection between boundaries of the containing regions and the sweeping line L. The set of line segments (boundaries of the containing regions) during the sweep are placed in a balanced binary search tree. This enable us to keep track of the vertical ordering of the line segments. Also, we can insert, delete and find a line segment in logarithmic time. At the left (right) endpoint of a line segment of a containing region, we add (delete) that segment into the tree. We also need to test whether two adjacent segments are intersecting, if so, their intersection point is added into the queue.

Each node of the tree contains a line segment. We also need to store in the tree the information needed to count the number of regions in which a point lies. For each edge of the tree an integer is stored in order to determine the number of regions that are being crossed when going down on that edge. Also, the root of the tree contains the total number of intact squares lying inside the polygon when its origin lies on the intersection point between the sweep line and the line segment being at the root. Figures 5.25 and 5.26 show examples.

Refer to Figure 5.25. Let q_i be the intersection point between the Sweep Line L and Boundary i. Let $R_{i,j}$ be the region with Boundaries i and j. $R_{i,j}$ is known as a containing region of a grid square. Walking down from q_5 to q_2 means that the total number of intact squares inside P does not change (actually we lost one square by leaving Region $R_{4,5}$, and we gain one square when entering $R_{2,3}$). However, walking up to q_4 means that the number of squares increases by one. Similarly, going down on L to reach q_1 means that the number of squares decreases. As shown in the figure, q_5 lies inside two regions ($R_{1,5}$ and $R_{4,7}$), while q_3 and q_4 lies inside three regions ($R_{1,5}$, $R_{2,3}$ and $R_{4,7}$).

The number of squares lying inside P at an intersection point in the diagram can be found in $O(\log(m+h+w))$ time for convex polygons and in $O(\log(m+r(h+w)))$ time for nonconvex polygons. This is due to the fact that there are at most $O(m+h+w)$ and $O(m+r(h+w))$ boundaries intersecting the sweep line L at a given x-coordinate, for convex and nonconvex polygons, respectively.

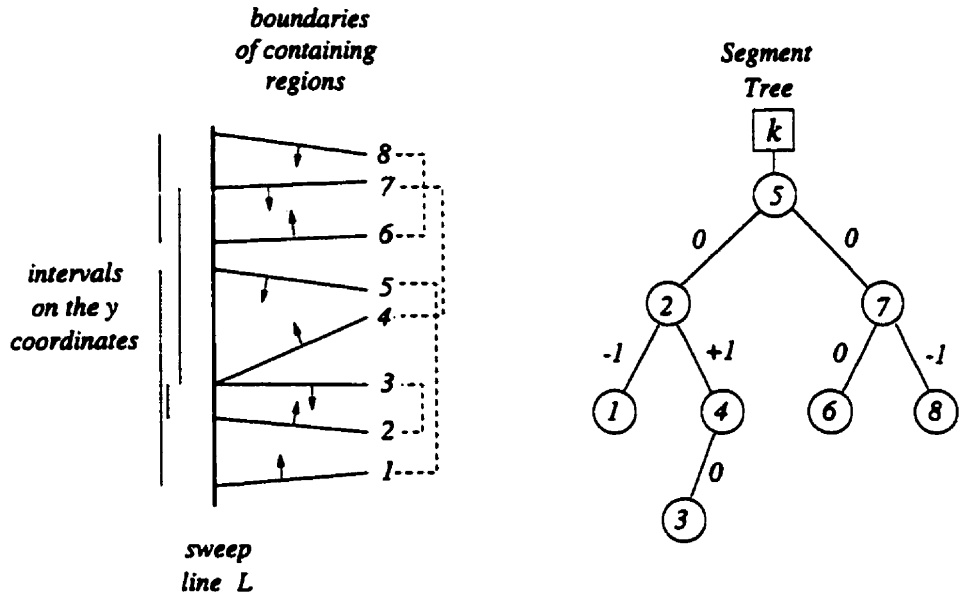

Figure 5.25: Segment tree of a set of boundaries of containing regions. Dashed lines show two line segments being part of the boundary of a given region.

5.2.7 Number of Intersection Points in the Diagram

In order to do a complexity analysis of Algorithm A2 we need the number of intersection points that are visited during the line sweep. For each point, we have to spend $O(\log(m+h+w))$ and $O(\log(m+r(h+w)))$ time for convex and nonconvex polygons respectively.

Convex Polygons

We know that the number of containing regions for convex polygons is $O(h+w)$. Thus, the total number of edges forming their boundaries is $O(m+h+w)$. Since line segments appearing inside the region comes from a grid point, most of the segments forming the boundaries are the sides of the unit square. However, vertices of P also hit grid points. They may produce a vertex inside $\mathcal{D}$, as shown in Figure 5.17 (containing region of Square h). The number of these vertices is m. Thus the total number of vertices in all containing regions is less than $m+5n$, where n is the number of containing regions (i.e. squares being hit by ∂P). As shown in Figure 5.17, Squares

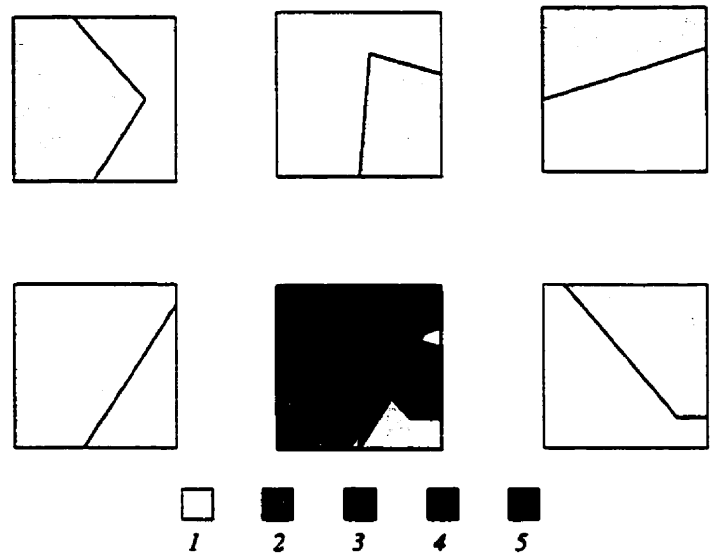

Figure 5.26: Translation diagram of five containing regions. An optimal placement has 5 squares inside the polygon. And it lies in the darkest region.

a and c have containing regions with 5 vertices.

An edge of P with k nearest neighbors produces $O(k)$ parallel line segments in the diagram. Thus, only line segments produced by different edges of P intersect. In the worst case this is quadratic because m may be small but h and w may be large. Figure 5.27 shows a convex polygon and its diagram. The number of intersection points is $O(h^2)$, since each line segment produced by one edge intersects $O(h)$ other line segments. Therefore, the total number of intersection points in the diagram is $O(m + h^2 + w^2)$, in the worst case.

Nonconvex Polygons

The total number of squares to consider when computing the containing regions is $O(r(h + w))$. As in the convex case, the worst case occur when line segments in the diagram produced by an edge e of P intersect other line segments produced by the other edges of P. For an initial placement of P on the grid, we need to know how many grid squares its boundary intersects when we translate P to the right only by at most one unit. In fact, $O(r(h + w))$ will be hit. If we translate the polygon

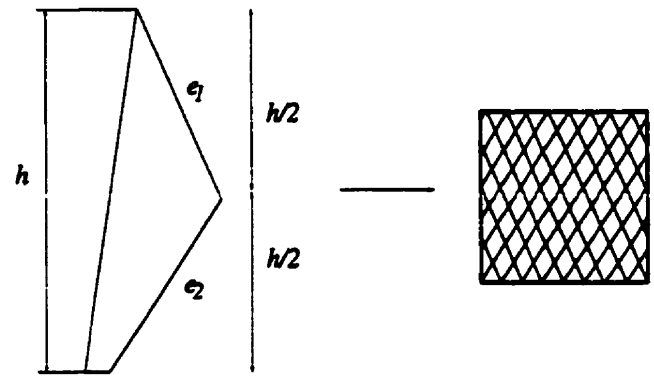

Figure 5.27: A convex polygon yielding $O(h^2)$ intersection points in its translation diagram.

in the y direction, its boundary will hit the same number of squares. Therefore, the number of intersection points inside the translation diagram $O(m + r^2(h^2 + w^2))$. $O(m)$ vertices are created by an intersection between a square and two adjacent edges of P, as explained in Section 5.2.5.

5.2.8 Worst Case Analysis

This section discusses the running time and the correctness of Algorithm A2. The next theorem shows that our algorithm finds an optimal placement.

Theorem 5.6 *Algorithm A2 finds an optimal placement to Problem 2.*

Proof. Algorithm A2 is a sweep line method that scans a translation diagram to find a point of maximum depth. By Theorem 3.3, that point corresponds to a placement where the polygon P has at least two contacts. Our algorithm tests only such points, which correspond to an intersection between at least two boundaries of containing regions. By Lemma 5.1, we know that these regions are correctly computed. Since our sweep line visits all intersection points, the correctness of our algorithm follows. ∎

The complexity analysis of Algorithm A2 is given by the following two theorems.

Theorem 5.7 *It takes $O\left((m+h^2+w^2)\log(m+h+w)\right)$ time to find an optimal solution to Problem 2, when the polygon is convex and has m edges.*

Proof. Computing the set of nearest neighbors takes $O(h+w)$ time because there are $O(h+w)$ such points. Computing the containing regions forming the translation diagram takes $O(m+h+w)$ time, as shown in Section 5.2.4. The next step is to count the number of intact squares inside the polygon from an initial placement. This takes $O(\min(h,w))$ time. Computing the Event Queue takes $O((m+h+w)\log(m+h+w))$ time because there are $O(m+h+w)$ line segments and P is convex. Applying a line sweep algorithm takes $O\left((m+h^2+w^2)\log(m+h+w)\right)$ time since $O(\log(m+h+w))$ time is spent to insert and delete in the binary search tree and the priority queue, for each intersection point. The size of the tree and the queue is $O(m+h+w)$, at any time during the line sweep. ∎

Theorem 5.8 *Given a nonconvex polygon with m edges and r reflex vertices, we can find in $O\left(m(h+w)+(m+r^2(h^2+w^2))\log(m+rh+rw)\right)$ time an optimal solution to Problem 2.*

Proof. Computing the set of nearest neighbors takes $O(r(h+w))$ time because there are $O(r(h+w))$ such grid points. The containing regions forming the translation diagram can be computed in $O(m(h+w))$ time. The number of intact squares inside the polygon from an initial placement can be found in $O(hl)$ time. Initializing the Event Queue takes $O((m+r(h+w))\log(m+r(h+w)))$ time because the line sweep intersects at most $O(m+r(h+w))$ line segments. Applying a line sweep algorithm takes $O\left((m+r^2(h^2+w^2))\log(m+rh+rw)\right)$ time since $O(\log(m+r(h+w)))$ time is spent to insert and delete in the binary search tree and the priority queue. The size of the tree and the queue is $O(m+r(h+w))$, at any time during the line sweep. This gives an algorithm running in $O\left(m(h+w)+(m+r^2(h^2+w^2))\log(m+rh+rw)\right)$ time. ∎

Note that if m is smaller than h or w, then the complexity of Algorithm A2 is $O(r^2(h^2+w^2)\log(rh+rw))$.

Chapter 6

Conclusion and Open Problems

In this thesis we focused on polygon placement problems. We have shown how to find an optimal placement of a simple polygon P on a grid of unit squares. The optimization criteria was the number of intact squares from the grid lying inside P. We designed efficient geometric algorithms and data structures. We have shown that we can visit a finite number of solutions, even if the number of possible placements of P on a grid is infinite. The complexity of our methods depends on the number of vertices of the polygon but also on its shape (height, width, number of reflex vertices and local minima). If P is nonconvex and has $O(1)$ number of reflex vertices, or does not have any local minima, then the complexity behaves as if the polygon was convex. The following table shows the list of algorithms and their complexity that have been discussed in this thesis.

Algorithm	Polygon	Complexity
A1	Convex	$O(m + h \log m)$
A1	Simple	$O((m + hl) \log m)$
A2	Convex	$O((m + h^2 + w^2) \log(m + h + w))$
A2	Simple	$O(m(h + w) + (m + r^2(h^2 + w^2)) \log(m + rh + rw))$

Table 6.1: Algorithms designed and their worst case analysis. The polygon has m edges, height h and width w. If the polygon is simple it has l local minima and a total of r reflex vertices.

Note that if $m = O(h+w)$ then Algorithm A2 takes $O(r^2(h^2+w^2)\log(rh+rw))$ time. However, if $h+w = O(m)$ then A2 runs in $O(r^2m^2\log(rm))$.

Although we studied the problem with a grid with unit squares, our methods can be applied to other grids with periodic patterns. We also studied the problem where grid points need to be sorted by their horizontal distances from a line segment e. These points all lie on one side of e. Our sorting algorithm runs in $O(n)$ time where n is the number of grid points. There is no restriction on that segment. Its endpoints may not be on grid points. Our sorting method was necessary to solve the problem where only x-translations are allowed.

The following are some open problems related to this research.

1. Given a set S of non overlapping squares in the plane and a simple polygon P, find a placement of P on the plane such that the number of squares from S inside P is maximum. Report the placement and the corresponding number of squares.

2. Given a simple polygon Q and an integer k, find a set of non overlapping squares of size at least k with minimum total area which covers entirely Q.

3. Design faster algorithms to solve problems discussed in the thesis.

Currently, we are investigating how to generalize oue methods if rotations are also allowed.

Bibliography

[AAS98] AGARWAL, P.K., AMENTA, N., and SHARIR, M. (1998), *Largest Placement of One Convex Polygon inside Another*, Discrete and Computational Geometry, v.19, n.1, pp. 95–104.

[AS96] AGARWAL, P.K. and SHARIR, M. (1996), *Efficient Randomized Algorithms for Some Geometric Optimization Problems*, Discrete and Computational Geometry, v.16, pp. 317–338.

[AS95] AGARWAL, P.K. and SHARIR, M. (1995), *Algorithmic Techniques for Geometric Optimization*, Lecture Notes in Computer Science, v.1000, pp. 234–253.

[Aga91] AGARWAL, P.K. (1991), *Intersection and Decomposition Algorithms for Planar Arrangements*, Cambridge University Press, Cambridge.

[AIUU96] AKIYAMA, J., ISHIGAMI, Y., URABE, M., and URRUTIA, J. (1996), *On Circles Containing the Maximum Number of Points*, Discrete Mathematics, v.151, n.1, pp. 15–18.

[AHS97] ALT, H., HSU, D., and SNOEYINK, J. (1997). *Computing the Largest Inscribed Isothetic Rectangle*, Proceedings of the Seventh Canadian Conference on Computational Geometry, pp. 67–72.

[AB89] AVNAIM, F. and BOISSONNAT, J.D. (1989), *Polygon Placement Under Translation and Rotation*, Theoretical Informatics and Applications, v.23, n.1, pp. 5–28.

[AB87] AVNAIM, F. and BOISSONNAT, J.D. (1987), *Simultaneous Containment of Several Polygons*, Proceedings of the 3rd Annual Symposium on Computational Geometry, pp. 242–250.

[BFM86] BAKER, B.S., FORTUNE, S.J., and MAHANEY, S.R. (1986), *Polygon Containment under Translation*, Journal of Algorithms, v.7, pp. 532–548.

[BB96] BAR-YEHUDA, R. and BEN-HANCOH, E. (1996), *A Linear-Time Algorithm for Covering Simple Polygons with Similar Rectangles*, International Journal on Computational Geometry and Applications, v.6, pp. 79–102.

[BDP95] BAREQUET, G., DICKERSON, M., and PAU, P. (1995), *Translating a Convex Polygon to Contain a Maximum Number of Points*, Journal of Combinatorial Theory – Series B, v. 18, pp. 39–41.

[BDK+96] BERG, M. DE, DEVILLERS, O., KREVELD, M.V., SCHWARKOPF, O., and TEILLAUD, M. (1996), *Computing the Maximum Overlap of Two Convex Polygons Under Translations*, Lecture Notes in Computer Science, v.1178, pp. 126–135.

[BKOS97] BERG, M. DE, KREVELD, M.V., OVERMARS, M., and SCHWARKOPF, O. (1997), *Computational Geometry - Algorithms and Applications*, Springer-Verlag, Berlin.

[BY98] BOISSONNAT, J.-M. and YVINEC, M. (1998), *Algorithmic Geometry*, Cambridge University Press, Cambridge.

[BB87] BRASSARD, G. and BRATLEY, P. (1987), *Algorithmique, Conception et Analyse*, Les Presses de l'Université de Montréal, Canada.

[CM91] CHASSERY, J.M. and MONTANVERT, A. (1991), *Géométrie Discrète en Analyse d'Images*, Hermès, Paris.

[Cha83] CHAZELLE, B. (1983), *The Polygon Containment Problem*, in PREPARATA, F.P., editor, *Advances in Computing Research, Volume 1, Computational Geometry*, JAI Press inc., London, pp. 1–32.

[CW90] CHIU, J.S. and WANG, J.S. (1990), *Monotone Polygon Containment Problem Under Translation*, Journal of Information Processing, v.13, n.4, pp. 486–493.

[CLR90] CORMEN, T.H., LEISERSON, C.E., and RIVEST, R.L. (1990), *Introduction to Algorithms*, MIT Press, Cambridge.

[CR94] CULBERSON, J.C. and RECKHOW, R.C. (1994), *Covering Polygon is Hard*, Journal of Algorithms, v.17, pp. 2–44.

[Dan95] DANIELS, K. (1995), *Containment Algorithms for Nonconvex Polygons with Applications to Layout*, Ph. D. thesis, Harvard University, Cambridge.

[DM96] DANIELS, K. and MILENKOVIC, V. (1996), *Distance-Based Subdivision or Translational LP Containment*, Proceedings of the Eighth Canadian Conference on Computational Geometry, pp. 196–201.

[DM95] DANIELS, K. and MILENKOVIC, V. (1995), *Multiple Translational Containment: Approximate and Exact Algorithms*, Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 205–214.

[DS98] DICKERSON, M. and SCHARSTEIN, D. (1998), *Optimal Placement of Convex Polygons to Maximize Point Containment*, Computational Geometry, Theory and Applications, vol. , n. 11, pp. 1–16.

[FH94] FISCHER, P. and HÖFFGEN, K.-U. (1994), *Computing a Maximum Axis-Aligned Rectangle in a Convex Polygon*, Information Processing Letters, v. 51, pp. 189–193.

[Fis93] FISCHER, P. (1993), *Finding Maximum Convex Polygons*, Lecture Notes in Computer Science, v.710, pp. 234–243.

[For85] FORTUNE, S.J. (1985), *A Fast Algorithm for Polygon Containment by Translation*, Lecture Notes in Computer Science, v.194, pp. 189–198.

[Gho90] GHOSH, P.K. (1990), *A Solution of Polygon Containment, Spatial Planning, and Other Related Problems Using Minkowski Operations*, Computer Vision, Graphics and Image Processing, v.49, pp. 1–35.

[GO97] GOODMAN, J.E. and O'ROURKE, J., editors (1997), *Handbook Discrete and Computational Geometry*, CRC Press, Boca Raton, New York.

[GC97] GRINDE, R.R. and CAVALIER, T.M. (1997), *A New Algorithm for the Two-Polygon Containment Problem*, Computers and Operations Research, v.24, n.3, pp. 231–252.

[GC96] GRINDE, R.R. and CAVALIER, T.M. (1996), *Containment of a Simple Polygon Using Mathematical Programming*, European Journal of Operational Research, v.92, pp. 368–386.

[GLS88] GRÖTSCHEL, M., LOVÁSZ, L., and SCHRIJVER, A. (1988), *Geometric Algorithms and Combinatorial Optimization*, Springer-Verlag, Berlin.

[GS89] GRÜNBAUM, B. and SHEPHARD, G.C. (1989), *Tilings and Patterns: An Introduction*, W.H. Freeman and Company, New York.

[Hoc97] HOCHBAUM, D., editor (1997), *Approximation Algorithms for NP-Hard Problems*, PWS Publication, Boston.

[KK99] KEATING, K. and KING, J. (1999), *Signed Tilings with Squares*, Journal of Combinatorial Theory – Series A, v.85, n.1, pp. 83–91.

[Kei97] KEIL, M. (1997), *Covering Orthogonal Polygons with Non-Piercing Rectangles*, International Journal on Computational Geometry and Applications, v.7, pp. 473–484.

[Ken97] KENYON, R. (1997), *Tilings of Convex Polygons*, Annales de l'Institut Fourier, v.47, pp. 929–944.

[Ken96] KENYON, R. (1996), *Tiling a Rectangle with the Fewest Squares*, Journal of Combinatorial Theory – Series A, v.76, n.2, pp. 272–291.

[LG97] LEVCOPOULOS, C. and GUDMUNDSSON, J. (1997), *Approximation Algorithms for Covering Polygons with Squares and Similar Problems*, Lectures Notes in Computer Science, v.1269, pp. 27–42.

[Man89] MANBER, U. (1989), *Introduction to Algorithms*, Addison-Wesley, New York.

[MS88] MARTIN, R.R. and STEPHENSON, P.C. (1988), *Putting Objects in Boxes*, Computer Aided Design, v.20, n.9, pp. 506–514.

[Mil98a] MILENKOVIC, V. (1998), *Rotational Polygon Containment and Minimum Enclosure*, Proceedings of the 14th ACM Symposium on Computational Geometry.

[Mil98b] MILENKOVIC, V. (1998), *Rotational Polygon Overlap Minimization and Compaction*, Computational Geometry: Theory and Applications, v.10, pp. 305–318.

[MD95] MILENKOVIC, V. and DANIELS, K. (1995), *Translational Polygon Containment and Minimal Enclosure using Geometric Algorithms and Mathematical Programming*, Technical Report TR-25-95, Center for Research in Computing Technology, Harvard University, Cambridge.

[MDL92] MILENKOVIC, V., DANIELS, K., and LI, Z. (1992), *Placement and Compaction of Nonconvex Polygons for Clothing Manufacture*, Proceedings of the Fourth Canadian Conference on Computational Geometry, pp. 236–243.

[MDL91] MILENKOVIC, V., DANIELS, K., and LI, Z. (1991), *Automatic Marker Making*, Proceedings of the Third Canadian Conference on Computational Geometry, pp. 243–246.

[Mit97] MITCHELL, J. (1997), *Approximation Algorithms for Geometric Optimization Problems (extended abstract)*, Proceedings of the Ninth Canadian Conference on Computational Geometry, pp. 229–232.

[Mor98] MORET, B.M. (1998), *The Theory of Computation*, Addison-Wesley, Reading, Massachusetts.

[Mul94] MULMULEY, K. (1994), *Computational Geometry - An Introduction Through Randomized Algorithms*, Prentice Hall, Englewood Cliffs, NJ.

[O'R98] O'ROURKE, J. (1998), *Computational Geometry in C*, Cambridge University Press, Cambridge.

[PA95] PACH, J. and AGARWAL, P.K. (1995), *Combinatorial Geometry*, John Wiley & Sons, Inc., New York.

[Pac93] PACH, J. (1993), *New Trends in Discrete and Computational Geometry*, Springer-Verlag, Berlin.

[PAB98] PAREDES, R., ARAGÓN, J.L., and BARRIO, R.A. (1998), *Nonperiodic Hexagonal Square-Triangle Tilings*, Physical Review B, v.58, n.18, pp. 11990–11995.

[PS85] PREPARATA, F.P. and SHAMOS, M.I. (1985), *Computational Geometry, an Introduction*, Springer-Verlag, New York.

[RHHB98] RICHARD, C., HÖFFE, M., HERMISSON, J., and BAAKE, M. (1998), *Random Tilings: Concepts and Examples*, Journal of Physics A, Mathematical and General, v.31, n.30, pp. 6385–6408.

[ST94] SHARIR, M. and TOLEDO, S. (1994), *Extremal Polygon Containment Problems*, Computational Geometry, Theory and Applications, v.4, pp. 99–118.

[Sir92] SIRBU, A. (1992), *Some Tilings Problems*, Tensor, v.54, pp. 174–178.