FINDING A NONREDUNDANT COMPONENT IN A POLYGON

by

Bradley Coleman

B.S., Rutgers University, 2003

Masters Thesis submitted in partial fulfillment
of the requirements for the degree of
Masters of Science
in the
School of Computing Science
Faculty of Applied Sciences

SIMON FRASER UNIVERSITY
Fall 2011

APPROVAL

Name: Bradley Coleman

Degree: Masters of Science

Title of Masters Thesis: Finding a nonredundant component in a polygon

Examining Committee: Dr. Andrei Bulatov,
Professor, Computing Science
Simon Fraser University
Chair

Dr. Tom Shermer,
Professor, Computing Science
Simon Fraser University
Senior Supervisor

Dr. Funda Ergun,
Professor, Computing Science
Simon Fraser University
Supervisor

Dr. Binay Bhattacharya,
Professor, Computing Science
Simon Fraser University
Examiner

Date Approved: 15 December 2011

Declaration of
Partial Copyright Licence

Abstract

Let v_i be a reflex vertex (internal angle greater than π) of a polygon P with n vertices. Extend the clockwise edge of v_i as a ray until it hits P, and then walk clockwise from v_i to the hitpoint. The chain we walked defines the clockwise component of v_i (it also has a counterclockwise component).

In $O(n)$ time we find some component of P that does not entirely contain another component, without using a general $O(n)$ time triangulation algorithm. This time bound has already been achieved using such a triangulation algorithm, but we show it is possible without it.

Our central algorithm simultaneously walks a component in the clockwise and counterclockwise directions. In these walks, it shoots rays and finds acceptable hitpoints that are not necessarily correct. For a particular hitpoint, the algorithm either validates it, disqualifies it and finds another, or shoots a new ray and finds a finds a new hitpoint.

Keywords: Polygon, Visibility, Computational Geometry, Linear-time.

Dedication

To mom and dad.

Acknowledgements

Many thanks to Tom Shermer. Also, to Hossein Jowhari, Rebecca Vossepoel, and Craig Weidert for math help. Thanks to Funda Ergun, Gerdi Snyder, Steve Pearce, Iman Hajirasouliha, Alexander Zaganas, Erez Maharshak, and Mark Latham for advice and support. And to mom and dad.

Contents

Approval ii

Abstract iii

Dedication iv

Acknowledgements v

Contents vi

List of Figures viii

List of Algorithms xiii

Index of notation and terminology xiv

1 Introduction 1

1.1 Algorithm overview 2

1.2 General definitions 3

1.3 Literature review 6

1.3.1 Art gallery problem 6

1.3.2 Watchman routes 7

1.3.3 LR-visibility 7

1.3.4 Weakly visibile edges and chords 8

1.3.5 Two-guard walks and room searches 9

1.4 Abstract data type for algorithms 9

2 Algorithm to find reflex-minima-free component **11**
2.1 Algorithm overview 14
2.2 An example 16
2.3 The algorithm 23
2.4 Preliminary lemmas 28
2.5 If D_h is non-interfering, then D_{h+1} is also non-interfering 36
2.5.1 Cutting crossings with $con(P_h)$ maintains non-interfering invariant . . 36
2.5.2 Cutting off a miss maintains non-interfering invariant 44
2.5.3 The algorithm maintains our non-interfering invariant 54
2.6 Correctness and running time analysis 55

3 Algorithm to find nonredundant component **59**
3.1 Obtain trapezoidation of reflex-minima-free polygon 60
3.2 Shortest paths, shortest path trees, and order queries 62
3.3 Find cw-nonredundant subcomponent in a reflex-minima-free component . . . 64
3.4 Algorithm to obtain nonredundant component from a polygon in $O(n)$ time . 66

4 Future work **67**

Bibliography **68**

List of Figures

1.1 The simple polygon P is in general position and has 12 vertices, labeled $v_0, v_1, \ldots, v_{11}$. The point p is a point, and v_0 is a point and a vertex. The polygon P is a Jordan Curve, and its interior is shaded and the exterior is not. Note that v_3 is not in the interior of P. The segment v_3v_4 is an edge, and the closed chain $P[v_3, v_8]$ includes v_3 and v_4, but not v_2. The vertex v_3 is convex and v_8 is a reflex vertex. 4

1.2 The segment v_pp has two crossings with P, the points a and b. The points v_p and p are not crossings because their neighborhoods only contain part of v_pp that is in the interior of P, not the exterior as well. The segments pa and bv_p are chords, and ab is not. 5

1.3 The ray from v_p through v'_p is the clockwise ray of v_p, denoted $\overrightarrow{cw}(v_p)$. The point v'_p is the hit point of this ray. Similarly, the counterclockwise hitpoint of $\overrightarrow{ccw}(v_j)$ is v''_j. The clockwise component of v_p, denoted $cw(v_p)$, is dotted and is the chain $P[v'_p, v_p]$. The counterclockwise component, $ccw(v_p)$, is the chain $P[v_p, v''_p]$. The component $ccw(v_q)$ is nonredundant, while $cw(v_p)$ and $ccw(v_p)$ are redundant because they both contain $ccw(v_q)$. 5

2.1 Each of the three chains $P[\alpha_1, w_h]$, $P[\alpha_2, w_h]$, and $P[\alpha_3, w_h]$ are candidate cw-components. Arbitrarily let $\alpha_3 = w^*_h$. Thus, the chain $D_h = P[w_h, w^*_h]$, and $P_h = P[w^*_h, w_h]$. The candidate cw-component P_h in standard position. Since $w^*_h \neq w'_h$, we know that P_h is not a component. The segment $con(P_h) = w^*_h w_h$ and this segment is a pseudochord. 12

2.2 The candidate cw-component P_h is soild and D_h is dashed. The chain D_h is balanced, is not ext-right-heavy, and $con(P_h)$ is a pseudochord. Thus D_h is non-interfering. 13

2.3 The candidate cw-component P_h is soild and D_h is dashed. The chain D_h is balanced, is ext-right-heavy, and $con(P_h)$ is a pseudochord. Thus D_h is interfering. 14
2.4 The function `getReflexMinFreeComp` chooses any reflex vertex v_i, and then finds the hitpoint of its clockwise ray v_i' by brute force. The chain $P[v_i', v_i]$ is $cw(v_i)$. This component is also known as P_0: the vertex v_i becomes w_0, and v_i' becomes w_0^*. The component P_0 is stored in `W` and then passed to `scanner`. The component P_0 is soild and D_0 is dashed. 17
2.5 The clockwise and counterclockwise walks of `scanner` start from w_0 and w_0^*, respectively. 18
2.6 The clockwise and counterclockwise walks of `scanner` halt when the clockwise walk finds a cw-miss v_j. The vertex v_j is a cw-miss since it is the first time the clockwise walk from w_0 has encountered more reflex-minima vertices than convex maximas. Next `scanner` calls `cwMissChoose` which will either call `cwSegCut` or `cwRayCut`. In the next figure v_j becomes w_1 and we search for and then find w_1^*. 19
2.7 The function `cwMissChoose` walks $P[w_1, w_0]$ and finds the closest crossing between $\overrightarrow{cw}(w_1)$ and this chain, if it exists. Since there is such a crossing p, the function `cwMissChoose` calls `cwSegCut`. The function `cwSegCut` walks counterclockwise from w_0^* until it crosses the segment pw_1. This first crossing becomes w_1^*. We delete the chains $P[w_1, w_0]$ and $P[w_0^*, w_1^*]$. 20
2.8 We now recurse on P_1. Again P_1 is solid and $D_1 = P[w_1, w_1^*]$ is dashed. The function `scanner` again begins its clockwise and counterclockwise walks from w_1 and w_1^*, respectively. The counterclockwise walk of `scanner` finds a crossing with $con(P_1)$ and calls `cwConnectorCut` since P_1 is a candidate cw-component. This function walks counterclockwise from w_1^* until finding a crossing that will become w_2^*. We delete the chain $P[w_1^*, w_2^*]$. 21
2.9 We now recurse on P_2. Again P_2 is solid and $D_2 = P[w_2, w_2^*]$ is dashed. The function `scanner` again begins its clockwise and counterclockwise walks from w_2 and w_2^*, respectively. This time both walks of `scanner` terminiate without event, thus P_2 has no reflex minima vertices and no crossings with $con(P_2)$, so it is a reflex-minima-free component. The algorithm outputs P_2. 22

2.10 The vertex v_j is a cw-miss, but `scanner` will not select it since it is not technically a reflex minima. 23
2.11 In both figures the chain $P[v_a, v_b]$ contains no reflex minima vertices, and only contains the convex maxima vertices v_a and v_b. However in the left figure, v_a is lower than v_b. 28
2.12 We partition P into P^1 and P^2 by shooting a ray straight down from v_q. . . . 30
2.13 The segment xy is a pseudochord in P, not a chord, because in the neighborhoods of both x and y, the segment is in the interior of P. 30
2.14 The shaded region is R. This is an example of when $w'_h \in D_h$. Observe that D_h is ext-right heavy with respect to $con(P_h)$ and thus it is interfering. . . . 32
2.15 This is the polygon P, that R in Figure 2.16 is based on. The function `scanner` will find the cw-miss v_j, where v_j is the lowest vertex in $R[v_j, w_h]$. 33
2.16 The chain $R[v_j, w_h]$ comes from an underlying polygon P, depicted in the above Figure 2.15. The polygon R is constructed around $P[v_j, w_h]$. Observe that the vertex v_j is a reflex minima vertex in P, but not in R. 33
2.17 The counterclockwise walk of `scanner` walks above the connector at the point p. We will show a contradiction, since the counterclockwise walk of `scanner` would have found the ccw-miss v_j before reaching p. 34
2.18 The counterclockwise walk of `scanner` walks above the connector at the point p. Then `scanner` sets `ccwWalkActive` to false. We will show that the clockwise walk of `scanner` will always find a cw-miss before crossing $w_h p$, in this case v_j. 35
2.19 The clockwise step of `scanner` will find the crossing p with $con(P_h)$. Since P_h is a candidate cw-component, it calls `cwConnectorCut`, which finds x and deletes $P[w^*_h, x]$. The vertex x becomes w^*_{h+1} and w_{h+1} will be w_h. 37
2.20 The interior of R is shaded. Note that `cwConnectorCut` would select x and not w'_h. However, w'_h is a satisfying crossing. 37
2.21 Observe that D_h is ext-right heavy with respect to $con(P_h)$ and thus interfering. Also observe that $P[z, w^*_h]$ is balanced, so it does not contain more ext-left crossings than ext-rights with respect to $w^*_h w^*_{h+1}$. 39
2.22 This figure shows that if y is an ext-right-crossing, then D_h is ext-right heavy with respect to $con(P_h)$, and thus D_h is interfering. 39
2.23 We will analyze the crossings of $P[y, w^*_h] = R'[w^*_h, y]$ with respect to both $w^*_h w^*_{h+1}$ and pq. 39

2.24 The polygons F and G are shaded. The chain $P[y, w^*_{h+1}] \subset R$ is drawn solid, the chain $P[z, y]$ is dashed, and the rest of P is dotted. The ext-left-crossing y is the first crossing with $con(P_{h+1})$ found in a clockwise walk of D_h from w^*_h, and z is the first crossing where D_h is ext-right heavy with respect to $con(P_{h+1})$. Note that $a, b, c, d \in R$, and that ab and cd do not cross R, however cd is a chord in R and ab is not. Finally, observe that $P[z, y]$ is balanced with respect to ab and cd, as well as $w^*_h w^*_{h+1}$. 40
2.25 The first event to occur is for the clockwise step of `scanner` to find the cw-miss v_j. It then calls `cwMissChoose` which finds p, the closest crossing with $\overrightarrow{cw}(v_j)$ to v_j in $P[v_j, w_h]$. It then calls `cwSegCut(W,`v_j`,p)` which traverses counterclockwise from w^*_h until it crosses the segment $v_j p$. However, before it does, it crosses $con(P_h)$ at y and ignores v_j and calls `cwConnectorCut` because P_h is a candidate cw-component. 45
2.26 This figure is used in a contradiction proof. We assume that `scanner` found the cw-miss v_j, even though it would have chosen the counterclockwise neighbor of v'_j instead. 45
2.27 This figure is used in a contradiction proof. The function `scanner` found the cw-miss v_j. Assume that $v'_j \in D_h$. 46
2.28 The clockwise step of `scanner` finds the cw-miss v_j. It calls `cwMissChoose` which calls `cwRayCut` which traverses counterclockwise from w^*_h until reaching y. Note that z was the first time that `crossingCounter == 1`, but at the time, it was not the closest known crossing to v_j. Thus, the walk continued until reaching y. Note that v'_j also satisfies the four conditions. 47
2.29 The function `cwRayCut` walks counterclockwise from w^*_h and chooses v'_j. 48
2.30 The subpolygon Q is shaded. We will consider v'_j to be an ext-left crossing in Q as it is in P. Note that the crossings with $\overrightarrow{cw}(v_j)$ that are closest (v'_j) and furthest from v_j (z) are ext-left crossings, and that there is one more ext-left crossing. 48
2.31 The first event to occur is for the clockwise step of `scanner` to find the cw-miss v_j. It then calls `cwMissChoose` which calls `cwSegCut(W,`v_j`,p)`, which traverses counterclockwise from w^*_h until crossing pv_j at y. Note that if the walk continued to v'_j, this would also satisfy the two conditions. 50

2.32 The polygon Q is shaded, and D_h is dashed. The point $p \in D_h$ is the first ext-right heavy crossing with respect to $con(P_{h+1})$ that we find in a clockwise walk from w_h^*, and q is the previous ext-right-crossing. 52
2.33 The chains P_h and D_h do not cross $con(P_h)$. The polygon A is shaded lighter than B. The chain D_h contains one more convex maxima than reflex minima, and P_h contains exactly as many reflex minimas as convex maximas. In this figure, P_h is not reflex-minima-free. The vertex v_k is a reflex minima and it's counterclockwise neighbor is a convex maxima. Note that the clockwise walk of `scanner` will not call a cutter, but the counterclockwise walk will upon reaching the ccw-miss v_k. 57

3.1 The rays $\overrightarrow{r_1}$, $\overrightarrow{r_2}$, and $\overrightarrow{s_4}$ are shot into the interior and $\overrightarrow{r_3}$ and $\overrightarrow{s_5}$ are not. . . . 63
3.2 The thick black path is $SP(q, v_t)$ 63
3.3 The thick black tree shows $SPT_P(q)$. That is, the shortest path tree of P rooted at q. Note that v_p is the parent of p and q is the parent of v_p in $SPT_P(q)$. 64

List of Algorithms

1 getReflexMinFreeComp(Chain V) 23
2 scanner(Chain W) 24
3 cwConnectorCut (Chain W) 25
4 cwMissChoose (Chain W, Vertex cwMiss) 25
5 cwRayCut (Chain W, Vertex cwMiss) 26
6 cwSegCut (Chain W, Vertex cwMiss, Point closestHit) 27
7 cwTrapezoidation (Chain W) 61
8 getCwNonRedundantComp (Chain W, Triangulation of W) 65
9 getNonRedundantComp (Chain V) 66

Index of notation and terminology

w_h^{**}, 12
w_h^*, 12
$\overrightarrow{pq}$, 5
v_i'', 5
v_i', 5

balanced, 13

candidate ccw-component, 12
candidate cw-component, 12
$ccw(v_i)$, 5
$\overrightarrow{ccw}(v_i)$, 5
ccw-component, 5
ccw-miss in P_h, 12
`Chain` (abstract data type), 9
chain, closed, 4
chain, open, 3
chord, 5
component, 5
component, ccw-nonredundant, 59
component, clockwise, 5
component, counterclockwise, 5
component, cw-nonredundant, 59
component, nonredundant, 6
component, redundant, 6
component, reflex-minima-free, 6
connector, 5
$con(P_h)$, 12
crossing, 4
cutter function, 16
$cw(v_i)$, 5
$\overrightarrow{cw}(v_i)$, 5
cw-component, 5
cw-miss in P_h, 12

edge, 3
ext-left crossing, 12
ext-right crossing, 12
ext-right heavy, 13

general position, 4

interfering, 13

n, 4
non-interfering, 13

order query, 64

P, 4
$P[p, q]$, 4
P_h, 12
parent in shortest path tree, 63
polygon, 4
polygon exterior, 4
polygon interior, 4
pseudochord, 12

ray, 5
ray, clockwise, 5
ray, counterclockwise, 5

$SP(p, q)$, 63
$SPT_P(q)$, 63
shortest path, 63
shortest path tree, 63
simple, 4
standard position, 6

`V`, 9
v_i, 3
vertex, 3
vertex, convex, 4
vertex, convex maxima, 6
vertex, reflex, 4
vertex, reflex minima, 6
visible (covisible), 5

`W`, 9
`W[x]`, 9

`Z.ccwEdge`, 10
`Z.ccwEnd`, 10
`Z.ccwNeigh`, 9
`Z.connector`, 10
`Z.cwEdge`, 10
`Z.cwEnd`, 10
`Z.cwNeigh`, 9
`Z.delete`, 10
`Z.setInitialCwComponent`, 10

Chapter 1

Introduction

In this thesis, we present an algorithm that finds any nonredundant component of a polygon with n vertices in $O(n)$ time. Let v_i be a reflex vertex (internal angle greater than π) of a polygon P with n vertices. Extend the edge immediately clockwise of v_i as a ray until it hits P, and then walk clockwise from v_i to the hitpoint. The chain we walked defines the clockwise component of v_i (it also has a counterclockwise component).

A component is nonredundant if it does not entirely contain another component. A chord is a segment that connects two boundary points of a polygon and lies entirely inside of the polygon.

This problem is already solved in the paper *LR-visibility in Polygons* [11] by Das, Heffernan, and Narasimhan. However, their algorithm uses Chazelle's complex linear-time triangulation algorithm [7] from 1991. These authors use [7] to build shortest path trees with the algorithm of Guibas, Hershberger, Leven, Sharir, and Tarjan [17].

Shortest path trees facilitate a powerful tool known as an *order query* which makes finding a single nonredundant component straight-forward. These authors use order queries for the problem of LR-visibility and finding weakly visible chords.

Their algorithm finds all of the nonredundant components of a polygon in $O(kn)$ time, where k is the number of disjoint nonredundant components. Since polygons with weakly visible chords, or that are LR-visible, have at most two disjoint nonredundant components, their algorithm runs in linear-time.

In addition to Chazelle's deterministic algorithm, there are also other very fast randomized triangulation algorithms. Also in 1991, Seidel published an algorithm with an expected running time of $O(nlog^*n)$. The function log^*n is so slow growing that this algorithm is

essentially linear. Then in 2000, Amato, Goodrich, and Ramos introduced a $O(n)$ expected running time algorithm for general polygons [1].

Despite these very fast generalb triangulation algorithms, complex and not complex, the goal of this thesis is to find any nonredundant component without the help of a general triangulation algorithm. To do so, we employ a novel technique.

1.1 Algorithm overview

We start with any component, and push the chord that connects both ends of the component further and further into this component. As we push this chord, what was a chord becomes a "pseudochord", and what was a component becomes a "candidate component". That is, this pseudochord may cross the polygon.

However, the crossings maintain an invariant. Particularly this invariant controls the crossings of the part of the polygon that is behind us as we walk into the original component, not in front of us. If the crossings of the chain behind us, with respect to the pseudochord, satisfy two natural conditions, a balance and an ordering, then this chain is said to be "non-interfering".

To handle the winding of P around and through this pseudochord, inspiration was gathered from the paper *A linear-time algorithm to remove winding of a simple polygon* by Bhattacharya, Ghosh, and Shermer [4].

We use the term pseudochord because the interior of P is locally in the neighborhood of the pseudochord's two endpoints. However, P might cross the pseudochord, so it is not necessarily a chord. Eventually, we will not be able to push it forward any more, and the chain in front of it will again become a component, and the pseudochord will again become a chord.

This algorithm consists of a simultaneous clockwise and counterclockwise walk from both ends of the pseudochord into the current candidate component.

Both walks are looking for either a special reflex vertex or a crossing between the pseudochord and the polygon. If a crossing is found, a function is called that advances one end of the pseudochord to a closer crossing, though not necessarily the closest. If a special reflex vertex is found, other functions are called that will either advance one or both ends of the pseudochord in a similar fashion. Note that the pseudochord does not sweep, over the polygon. More accurately, it jumps.

The component that is finally produced is not necessarily nonredundant, but it is "reflex-minima-free". This class of components is significant only because it is very easily triangulated in linear-time (algorithm detailed in Chapter 3). Chapter 2, simply details and proves an algorithm that takes as input any component and produces a reflex-minima-free subcomponent, if one exists.

Once we have a triangulated component, we have successfully avoided using a general triangulation algorithm. Then, we are free to employ techniques similar to [11] to reduce the component further. We run a simplified version of their algorithm, which is detailed in Chapter 3.

The result of applying this algorithm is a component that does not contain another clockwise component. We call this a "cw-nonredundant" component. Note that a cw-nonredundant component can be clockwise or counterclockwise.

Because the component is cw-nonredundant, does not mean it is reflex-minima-free. Thus, we again call our main algorithm on it to again produce a reflex-minima-free subcomponent, if one exists. Then again, we triangulate it as before.

Finally, we call the symmetric version of the same simplified version of [11] to produce a subcomponent, if one exists, that contains no counterclockwise components. Hence a "ccw-nonredundant" component.

Alas, the component is both cw-nonredundant and ccw-nonredundant and thus cannot contain any clockwise or counterclockwise components. Therefore it is nonredundant.

In Chapter 2, our most difficult task, by far, is to prove that the non-interfering invariant is maintained each time the pseudochord moves forward. We also must show that eventually the pseudochord becomes a chord that connects the endpoints of a reflex-minima-free component. Then we show that this all runs in linear-time. In Chapter 3 we show the running time and correctness of the triangulation algorithm, and the simplified version of [11].

1.2 General definitions

(See Figure 1.1) Given an ordered sequence of points $v_0, v_1, \ldots, v_{n-1}$, where $v_i \in \mathbb{R}^2$, the *open chain* on these points is the piecewise linear curve consisting of the ordered sequence of line segments $v_0v_1, v_1v_2, \ldots, v_{n-2}v_{n-1}$. The points v_i are called *vertices* and the line segments v_iv_{i+1} are called *edges*. We consider edges and line segments to be closed, so the edge or line segment pq contains p and q.

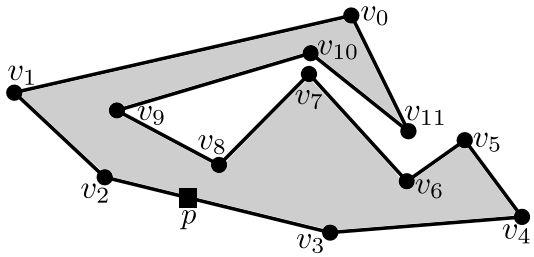

Figure 1.1: The simple polygon P is in general position and has 12 vertices, labeled $v_0, v_1, \ldots, v_{11}$. The point p is a point, and v_0 is a point and a vertex. The polygon P is a Jordan Curve, and its interior is shaded and the exterior is not. Note that v_3 is not in the interior of P. The segment v_3v_4 is an edge, and the closed chain $P[v_3, v_8]$ includes v_3 and v_4, but not v_2. The vertex v_3 is convex and v_8 is a reflex vertex.

(See Figure 1.1) A *closed chain* $v_0, v_1, \ldots, v_{n-1}$ forms a circular ordering and is composed of the open chain $v_0, v_1, \ldots, v_{n-1}$ and the edge $v_{n-1}v_0$. This edge is adjacent to both $v_{n-2}v_{n-1}$ and v_0v_1. For a closed chain, we will refer to vertices modulo n, so $v_n \equiv v_0$; this enables us to refer to any edge in a closed chain with v_iv_{i+1}. All chains will be assumed to be closed unless otherwise stated.

(See Figure 1.1) An open or closed chain is considered *simple* if non-adjacent edges have an empty intersection and no three consecutive vertices are collinear. A closed simple chain is a Jordan Curve and therefore divides the plane into three regions: the curve itself, an unbounded region called the *exterior*, and a bounded region called the *interior*.

(See Figure 1.1) A *polygon* is a closed simple chain with n vertices. The vertices of P will be numbered in increasing order in the counterclockwise direction, so the counterclockwise neighbor of v_i is v_{i+1}. Given an open chain or a polygon, if no three vertices are collinear and no four vertices can define two parallel lines, then it is said to be in *general position*. We will assume that polygons are in general position unless otherwise noted.

(See Figure 1.1) Given two points or vertices, $p, q \in P$, the notation $P[p, q]$ will refer to the open chain $p, v_i, v_{i+1}, \ldots, v_k, q$, where the vertex v_i is the vertex immediately counterclockwise of p and v_k is the vertex immediately clockwise of q.

(See Figure 1.1) Each vertex $v_i \in P$ has an interior angle and an exterior angle, both defined by the two edges that v_i is an endpoint of. If the interior angle of v_i is less than π, then it is a *convex vertex*, and if the interior angle is greater than π, it is a *reflex vertex*. Interior angles of π are not present in simple polygons.

(See Figure 1.2) A *crossing* of a line l (or ray, or segment) is a point $p \in P$ where in any

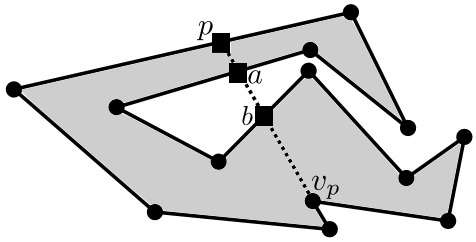

Figure 1.2: The segment $v_p p$ has two crossings with P, the points a and b. The points v_p and p are not crossings because their neighborhoods only contain part of $v_p p$ that is in the interior of P, not the exterior as well. The segments pa and bv_p are chords, and ab is not.

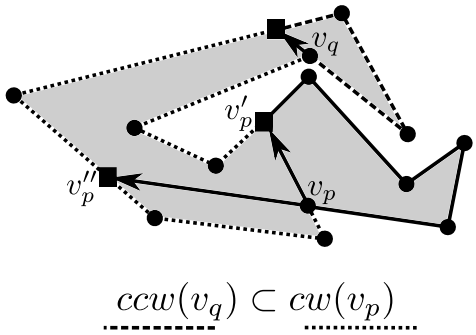

Figure 1.3: The ray from v_p through v'_p is the clockwise ray of v_p, denoted $\overrightarrow{cw}(v_p)$. The point v'_p is the hit point of this ray. Similarly, the counterclockwise hitpoint of $\overrightarrow{ccw}(v_j)$ is v''_j. The clockwise component of v_p, denoted $cw(v_p)$, is dotted and is the chain $P[v'_p, v_p]$. The counterclockwise component, $ccw(v_p)$, is the chain $P[v_p, v''_p]$. The component $ccw(v_q)$ is nonredundant, while $cw(v_p)$ and $ccw(v_p)$ are redundant because they both contain $ccw(v_q)$.

neighborhood of p, there is part of l that is in the interior of P and another part of l that in the exterior of P. If P does not cross pq, then this segment is a *chord*. The points p and q are also said to be *visible* or *covisible*.

(See Figure 1.3) The *ray* from p to q, denoted $\overrightarrow{pq}$, includes the segment pq and all points r where q is in the segment pr. Each reflex vertex $v_i \in P$ defines two rays: the *clockwise ray* $\overrightarrow{v_{i-1}v_i}$, denoted $\overrightarrow{cw}(v_i)$, and the *counterclockwise ray* $\overrightarrow{v_{i+1}v_i}$, denoted $\overrightarrow{ccw}(v_i)$. Of the crossings between $\overrightarrow{cw}(v_i)$ and P, the closest to v_i will be denoted as v'_i. Similarly, the point v''_i is the crossing between $\overrightarrow{ccw}(v_i)$ and P that is closest to v_i.

(See Figure 1.3) Each reflex vertex $v_i \in P$ defines two *components*. The *cw-component* of v_i, denoted $cw(v_i)$, is the chain $P[v'_i, v_i]$ and the *ccw-component* of v_i, denoted $ccw(v_i)$, is the chain $P[v_i, v''_i]$. The *connector* of $cw(v_i)$ is the segment $v'_i v_i$, and $cw(v_i)$ together with its connector, forms a polygon. The symmetric holds for $ccw(v_i)$ and its connector, $v_i v''_i$.

Note that because v_i' and v_i'' become vertices in $cw(v_i)$ and $ccw(v_i)$ respectively, these chains contain three collinear vertices, and are therefore not in general position. We will simply excuse this.

(See Figure 1.3) A component that does not entirely contain another component is considered to be *nonredundant*. A component that entirely contains another component is *redundant*. Nonredundant components are the central focus of this thesis.

All components will be considered to be in *standard position*. The connector will be horizontal. If the component is clockwise, for instance $cw(v_i)$, then v_i will be to the right of v_i', and the counterclockwise neighbor of v_i' will be below the connector. Similarly, if the component is counterclockwise, $ccw(v_j)$, then v_j will be left of v_j' and the clockwise neighbor of v_j' will be below the connector. Standard position enables us to use terms like *above*, *below*, *left* and *right*.

Consider a polygon in standard position about some component. A vertex $v_i \in P$ is a *reflex minima vertex* if it is reflex and both of its neighbors are above it, or it is a *convex maxima vertex* if it is convex and both of its neighbors are below it. A component is *reflex-minima-free* if it contains no reflex minima vertices.

1.3 Literature review

Components have been well studied in the literature. They form the foundation of many intuitive problems involving both motion planning inside of a polygonal environment, like the problem of finding watchman routes, and motion planning along the boundary of a polygon, like LR-visibility and the two-guard problem. In the following subsections we will attempt to demonstrate the usefulness and practicality of components and nonredundant components.

Let us first define weak visibility. Two sets are weakly covisible if each element of each set can see some element of the other.

1.3.1 Art gallery problem

Famously proposed by Klee in 1973, it asks how many stationary guards are necessary, so that together they can see every point in an art gallery. This problem is extremely natural and practical, particularly if guards are seen as sensors, or cell phone towers where there are mountains outside of the polygon to block the signal, or even Exxon gas stations.

Zhong Zhang, in his PhD thesis [34], points out that this problem has been studied so extensively that O'Rourke wrote a book about it [23]. Shermer also wrote a survey on the subject [27].

Besides being one of the seminal problems of polygonal visibility, of note is that there must be a guard stationed in each non-redundant component. Note that nonredundant components can overlap, so one guard may guard multiple nonredundant components.

As a sidebar, when formulated as a decision problem, Lee and Lin in [21] show that it is NP-hard. Chvátal in [9] and Fisk in [14] show that the upper bound of guards is $\lfloor n/3 \rfloor$. (In his elegant proof, Fisk triangulated P, three-colored the triangulation tree, picked one of the three colors arbitrarily, and then placed one guard in each triangle of that color.)

1.3.2 Watchman routes

A Watchman Route is the analog of the Art Gallery Problem, but as opposed to many stationary watchmen, there is one mobile watchman. He starts at a fixed point, and ends at this same point. A Watchman Route in a polygon P is a closed chain in P such that every point of P is visible from some point in the chain. In other words, the chain and P are weakly covisible. Note that this approach does not protect against a mobile intruder.

Of note is that for the same reason that a guard must be stationed in each nonredundant component of an art gallery, a mobile watchman must peek into each nonredundant component of the gallery he is patrolling as well.

This problem was introduced by Chin and Ntafos in [24] in 1986. Since then, there have been many papers on this subject such as [12, 22], and many broken papers as well. For instance, [18] pointed out flaws in [25] and [30] but then [18] was later also broken.

Although optimal Watchman Routes can be found in polynomial time, some variations deal with NP-completeness and optimization, unlike this thesis. We raise this problem, along with the art gallery problem, only because of their intuitive nature, and to show how naturally they are expressed in terms of nonredundant components.

1.3.3 LR-visibility

In an appendix to this thesis, we will provide a slightly simplified rewrite of *LR-visibility in Polygons* by Das, Heffernan, and Narasimhan [11]. Note that this paper is also detailed in Ghosh's book *Visibility Algorithms in the Plane* [16].

Two points $s, t \in P$, partition P into two subchains. Let these chains be called L and R. The polygon P is LR-visible about s and t if L and R are weakliy covisible.

In linear-time, the algorithm returns all pairs s and t, that P is LR-Visible with respect to. Note that it uses Chazelle's triangulation algorithm [7] as a preprocessing step.

In this paper the authors show that a polygon is LR-visible with respect to s and t if and only if each nonredundant component contains both s and t. Thus, if there are three disjoint nonredundant components, the polygon cannot be LR-visible.

The authors give an algoirthm that uses a kind of order query based on a shortest path tree generated by [17] and [7]. It finds all of the nonredundant components in P in $O(kn)$ time, where k is the number of disjoint nonredundant components. Since they are dealing with LR-visibility, they stop if k reaches 3, thus their algorithm runs in linear-time.

Therefore, their algorithm either finds all of the components of a polygon that has at least an LR-visible pair, or some of the components of a polygon that is not LR-visible. Note that this problem was solved essentially by finding nonredundant components. (See future work chapter.)

1.3.4 Weakly visibile edges and chords

In 1981, Avis and Toussaint gave a linear-time algorithm to determine if a polygon is weakly visible from an edge [2]. Note that a component is nonredudnant if and only if it is weakly visible from its connector. Sack and Suri later gave an algorithm that in linear-time computes all weakly visible edges of a polygon [26].

A natural extension of this problem is to study weakly visible chords. A chord is weakly visible, if it and P are weakly covisible.

Das, Heffernan, and Narasimhan in [10], find all weakly visible chords in linear-time, by calling their LR-visibility algorithm [11] as a subroutine. Thus, they are also using Chazelle's triangulation algortihm [7] to enable [17] to obtain the shortest path tree in linear-time. This paper leans heavily on their LR-visibility algorithm that finds nonredundant components.

Bhattacharya and Mukhopadhyay [5] find a single weakly visible chord, and all of the nonredundant components it intersects without using [7] or [17]. They also point out that polygons that have a weakly visible chord, are triangulatable in linear-time without a general triangulation algorithm.

Note that the problems of LR-visibility and finding weakly visible chords are related. To quote [10], "two points x and y of P are the endpoints of a weakly-visible chord of P if

and only if xy is a chord of P and P is LR-visible with respect to x and y." However, that is not to say that if P is LR-visible with respect to x and y, that xy is a chord.

1.3.5 Two-guard walks and room searches

In the Two-Guard problem, two guards start at the point $s \in P$ and walk in oppite directions along P while remaining covisible, until they both reach $t \in P$. The solution to this problem is a schedule for both guards.

Some variations have one guard starting at t and walking to s, while the other starts at s and walks to t. Others allow one guard to backtrack while the other works through an obstacle. Some focus on optimization.

In this context, components are obstacles that the guards, if possible, must navigate. Also note that If P is not LR-visibile with respect to s and t, then no schedule, regardless of the variation, can succeed. This problem is active, with Tan, Jiang, Zhang, Zhou, Ntafos publishing papers from 2006 to 2010: [32, 33, 31, 28, 35].

The Two-Guard Room Search problem is similar. Both guards start walking in different directions from one origin point on the polygon, called the *door*. Again they walk in opposite directions, but this time their objective is to let the chord between them sweep the entire polygon in an effort to catch an intruder hiding in the interior.

Again, components in this context are obstacles for the two searchers and hiding places for the intruder. This problem is also active with Kameda, Zhang, Tan, Bahun, Lubiw, Bhattacharya and Shi publishing papers from 2006 to 2010: [20, 29, 3, 6].

1.4 Abstract data type for algorithms

We create an *Abstract Data Type* (ADT) to store a chain or a polygon. This ADT is essentially an "object" in the parlance of modern programming languages. We will use this ADT in all of our algorithms.

We will store P permanently in the instance `v` and we will also make a copy, `w`, of `v` that we can edit without harming the original. This ADT supports the following operations, which we define for some instance `z`:

- `z.cwNeigh(x)`: returns the clockwise neighbor of `x`;
- `z.ccwNeigh(x)`: returns the counterclockwise neighbor of `x`;

- `Z.cwEdge(x)`: returns the clockwise edge of `x`, which is the edge `x,Z.cwNeigh(x)`;
- `Z.ccwEdge(x)`: returns the counterclockwise edge of `x`, which is the edge between x and `Z.ccwNeigh(x)`;
- `Z.delete(x,y)`: deletes the chain $P[x, y]$ from `Z` and possibly updates `Z.ccwEnd` and/or `Z.cwEnd`; and
- `Z.ccwEnd`: not defined if `Z` stores a polygon, otherwise return the clockwise endpoint of `Z`;
- `Z.cwEnd`: not defined if `Z` stores a polygon, otherwise return the counterclockwise endpoint of `Z`;
- `Z.connector`: not defined if `Z` stores a polygon, otherwise return the segment `Z.ccwEnd,Z.cwEnd`;
- `Z.setInitialCwComponent(ccwEnd, cwEnd)`: defined only if `Z` stores a polygon, it initializes and then returns a copy of the subset of `Z` that is a cw-component of a polygon.

This ADT will be named *Chain* and be implemented as a list. Thus, `Z` is an instance of the ADT Chain.

Chapter 2

Algorithm to find reflex-minima-free component

We will divide the problem of finding a single nonredundant component into subproblems. Let the given polygon P have n vertices. The first subproblem, and the subject of this chapter, is to find a reflex-minima-free component of P in $O(n)$ time.

The function `getReflexMinFreeComp`, which is Algorithm 1, takes `V` as input where `V` stores P, and finds a clockwise component $cw(v_x) \subset P$, and then initializes `W` so that it stores $cw(v_x)$.

This function walks clockwise from `W[0]` until it reaches some reflex vertex, v_i. If it does not find one, we know P has no components and we exit. Next it finds v_i' by checking each edge of `W` for an intersection with $\overrightarrow{cw}(v_i)$ and retaining the closest such intersection. Then it initializes `W` so that it stores exactly $cw(v_i)$. The vertex v_i becomes `W.ccwEnd`, the point v_i' becomes `W.cwEnd`, and then `getReflexMinFreeComp` calls `scanner(W)`.

The function `scanner` in Algorithm 2 recursively prunes `W` until it represents a reflex-minima-free component. We will denote the initial component found by `getReflexMinFreeComp` as P_0. The function `scanner` takes some P_h as input and then its helper functions delete part of P_h to form P_{h+1}, and then it recurses on P_{h+1}. If it cannot delete part of P_h, then P_h is reflex-minima-free. We will refer to the chain, $P \setminus P_h$, which the algorithm has deleted thus far, as D_h.

In order to stay within a linear time bound, we cannot insist that each P_h be a component. If `W` stores a chain that is not a component, then P crosses `W.connector`.

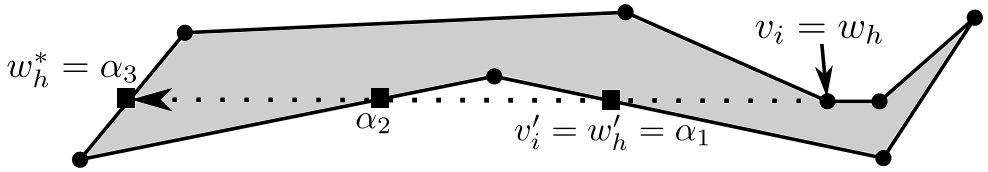

Figure 2.1: Each of the three chains $P[\alpha_1, w_h]$, $P[\alpha_2, w_h]$, and $P[\alpha_3, w_h]$ are candidate cw-components. Arbitrarily let $\alpha_3 = w_h^*$. Thus, the chain $D_h = P[w_h, w_h^*]$, and $P_h = P[w_h^*, w_h]$. The candidate cw-component P_h in standard position. Since $w_h^* \neq w_h'$, we know that P_h is not a component. The segment $con(P_h) = w_h^* w_h$ and this segment is a pseudochord.

(See Figure 2.1) Let v_i be a reflex vertex, and α be some crossing between P and $\overrightarrow{cw}(w_h)$. The chain $P[\alpha, w_h]$ will be called a *candidate cw-component*. Similarly, let β be some crossing between P and $\overrightarrow{ccw}(v_i)$. The chain $P[v_i, \beta]$ will be called a *candidate ccw-component*. Note that a component is also a candidate component, but a candidate component is not necessarily a component. Thus, the component P_0 and the final reflex-minima-free component are candidate components as well.

(See Figure 2.1) Let P_h be a candidate cw-component. The point w_h^* will denote the endpoint of P_h that is not v_i, and w_h will denote v_i itself. Similarly, if P_h is a candidate ccw-component, then w_h^{**} denotes the endpoint that is not v_i, and again w_h denotes v_i.

Candidate components also have a connector, which we will again denote as $con(P_h)$, that connects the two endpoints of P_h. If P_h is a candidate cw-component, $con(P_h) = w_h w_h^*$, and if it is a candidate ccw-component, $con(P_h) = w_h w_h^{**}$. Just like a component, a candidate component will be assumed to be in standard position about its connector.

Immediately after `scanner` recurses, `W` will store some candidate component P_h. If P_h is a candidate cw-component, then `W.ccwEnd` is w_h, `W.cwEnd` is w_h^* and `W.connector` is the segment $w_h w_h^*$. Similarly, if P_h is a candidate ccw-component, then `W.cwEnd` is w_h, `W.ccwEnd` is w_h^{**} and `W.connector` is the segment $w_h w_h^{**}$.

Let P_h be a candidate cw-component. The reflex vertex $v_j \in P_h$ will be called a *cw-miss in* P_h if $v_j' \in P[w_h^*, v_j]$ or a *ccw-miss in* P_h if $v_j' \in P[v_j, w_h]$. Otherwise, if P_h is a candidate ccw-component, then $v_j \in P_h$ is a cw-miss in P_h if $v_j' \in P[w_h, v_j]$ and a ccw-miss in P_h if $v_j' \in P[v_j, w_h^{**}]$.

Let P be in standard position about $\overrightarrow{cw}(v_i)$ and let $x \in P$ be a crossing with $\overrightarrow{cw}(v_i)$. If the exterior of P is locally to the right of x, then we will call this an *ext-right* crossing, otherwise it will be called an *ext-left* crossing.

(See Figure 2.1) A *pseudochord* is a segment xy, where $x, y \in P$ (note that x and y may

be points or vertices), and in the neighborhood of both x and y, the segment xy is in the interior of P.

(See Figures 2.2 and 2.3) A chain $P[a, b]$ will be called *ext-right heavy* with respect to a segment, ray or line if for some $c \in P[a, b]$, the chain $P[c, b]$ contains more ext-right crossings than ext-left crossings. A chain is *balanced* with respect to a segment, ray, or line if it has as many ext-left crossings as ext-right crossings.

(See Figures 2.2 and 2.3) The chain D_h is said to be *non-interfering* if $con(P_h)$ is a pseudochord, and if D_h is balanced and not ext-right heavy with respect to $con(P_h)$, and *interfering* otherwise. This is one of the central concepts of this chapter. Our main task will be to show that for each P_h, its corresponding D_h is non-interfering.

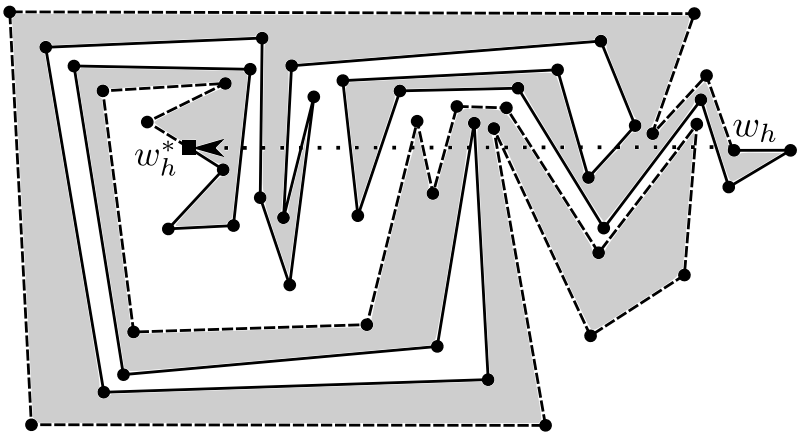

Figure 2.2: The candidate cw-component P_h is soild and D_h is dashed. The chain D_h is balanced, is not ext-right-heavy, and $con(P_h)$ is a pseudochord. Thus D_h is non-interfering.

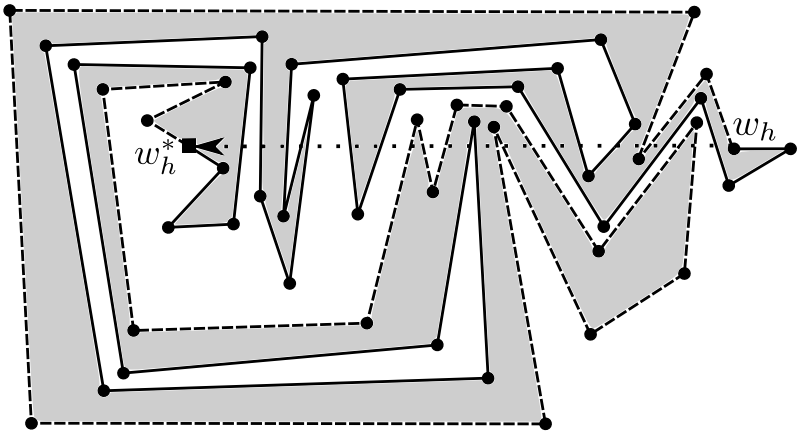

Figure 2.3: The candidate cw-component P_h is soild and D_h is dashed. The chain D_h is balanced, is ext-right-heavy, and $con(P_h)$ is a pseudochord. Thus D_h is interfering.

2.1 Algorithm overview

Before we proceed, note that we will frequently assume that P_h is a candidate cw-component, as opposed to a candidate ccw-component. We do this to avoid repeating an argument that is symmetrical, with the exception of different naming. For instance, w_h refers to `W.ccwEnd` if P_h is a candidate cw-component and `W.cwEnd` otherwise. We will generally not repeat an argument unless there is a material difference between the clockwise and counterclockwise cases.

The function `scanner` takes as input a component P_h and behaves equivalently if P_h is a clockwise or candidate ccw-component. It first checks the clockwise edge of `W.ccwEnd` and then the counterclockwise edge of `W.cwEnd` and then the clockwise edge of `W.cwNeigh(W.ccwEnd)`, and then the counterclockwise edge of `W.ccwNeigh(W.cwEnd)` and so on. We refer to these alternating walks as the clockwise and counterclockwise walks of `scanner`. It continues alternating like this until one of these three events occurs:

1. The function `scanner` terminates.
2. The clockwise or counterclockwise walk of `scanner` crosses `W.connector` (we treat both cases equivalently), or
3. The variables `cwExtremums` or `ccwExtremums` reach -1.
4. The function `scanner` crosses the line containing $con(P_h)$, but not $con(P_h)$ itself.

We will now explain how the algorithm handles these four events:

Case 1: scanner terminates

We know that `W` stores a reflex-minima-free component and we terminate.

Case 2: a walk of scanner crosses the connector

(See Figure 2.19) If P_h is a candidate cw-component, we call `cwConnectorCut` (see Algorithm 5), otherwise we call `ccwConnectorCut`. Without loss of generality, let P_h be a candidate cw-component. Now that `scanner` has discovered a crossing with $con(P_h)$, the point $w_h^{**} =$ `W.cwEnd` is no longer the closest known crossing to $w_h =$ `W.ccwEnd`.

The function `cwConnectorCut` walks counterclockwise from `W.cwEnd` looking for a crossing x with $con(P_h)$ such that: (A) x is the closest crossing to w_h that the walk has visited, (B) x is an ext-left-crossing, and (C) including x, the walk has visited as many ext-left-crossings as ext-right-crossings. It then deletes the chain that it walked. Lemma 7a shows that it will indeed find such a crossing. The crossing x still may not be the closest crossing with P_h and $con(P_h)$.

Case 3: `cwExtremums` or `ccwExtremums` reaches -1

If `cwExtremums` in `scanner` reaches -1 for $v_j \in P_h$, then Lemma 9a shows that v_j is a cw-miss in P_h. Similarly, if `ccwExtremums` reaches -1, then we have found a ccw-miss by Lemma 9b. If `scanner` finds a cw-miss it calls `cwMissChoose` (see Algorithm 4), and if it finds a ccw-miss, it calls `ccwMissChoose`.

Without loss of generality, let P_h be a candidate cw-component, and let v_j be the cw-miss found by `scanner`. The function `scanner` will then call `cwMissChoose` which simply determines if $\overrightarrow{cv}(v_j)$ intersects $P[v_j, w_h]$ or not. Note that since v_j is a cw-miss, $v'_j \notin P[v_j, w_h]$, but $\overrightarrow{cv}(v_j)$ can still intersect $P[v_j, w_h]$. If so, it finds q, the closest intersection to v_j, and calls `cwSegCut` (see Algorithm 6). Otherwise it calls `cwRayCut`. If `scanner` found a ccw-miss instead, `ccwMissChoose` searches $P[w_h, v_j]$ and calls either `ccwSegCut` or `ccwRayCut`.

The function `cwSegCut` walks counterclockwise from `W.cwEnd` until it crosses the segment qv_j. It then deletes everything it walked.

The function `ccwRayCut` also walks counterclockwise from `W.cwEnd`, but instead looks for a crossing x with $\overrightarrow{cw}(v_j)$ such that: (A) x is the closest crossing to v_j that the walk has visited, (B) x is an ext-left-crossing, and (C) including x, the walk has visited one more ext-left-crossing than ext-right-crossing. Then the chains $P[v_j, w_h]$ and $P[w_h^*, x]$ will be deleted from `W`. If v_j was a ccw-miss then $P[w_h, v_j]$ and $P[x, w_h^*]$ will be deleted form `W`.

(See Figure 2.25) If either `cwSegCut` or `cwRayCut` cross $con(P_h)$ before reaching a satisfying crossing, these functions will ignore the cw-miss v_j in P_h that they were called to handle. These functions will not delete anything from P_h and will instead call either `cwConnectorCut` or `ccwConnectorCut` which will delete part of P_h. It is essentially as if event (2) was triggered from the start, and event (3) was not.

If a crossing with $con(P_h)$ is not found, then Lemma 10a shows that `cwRayCut` will indeed find a satisfying crossing and Lemma 11a shows that `cwSegCut` will indeed cross qv_j.

Case 4: `scanner` crosses the line containing $con(P_h)$, but not $con(P_h)$ itself

If the clockwise walk of `scanner` walks above $con(P_h)$ without crossing $con(P_h)$, then `cwWalkActive` is set to false and the clockwise walk halts while the counterclockwise walk continues. Lemma 6a shows that the counterclockwise walk cannot also halt, and that instead it will find a ccw-miss, if it does not cross $con(P_h)$ first. The symmetric applies if the counterclockwise walk halts, then the clockwise walk will continue and find something to cut.

We will refer to `cwConnectorCut`, `cwRayCut`, `cwSegCut` and their counterclockwise counterparts as *cutters* because these functions alone remove parts of P_h.

2.2 An example

In the next five pages, we will show an example of the central algorithm of this chapter, `getReflexMinFreeComp` that takes as input a component, and outputs a reflex-minima-free component.

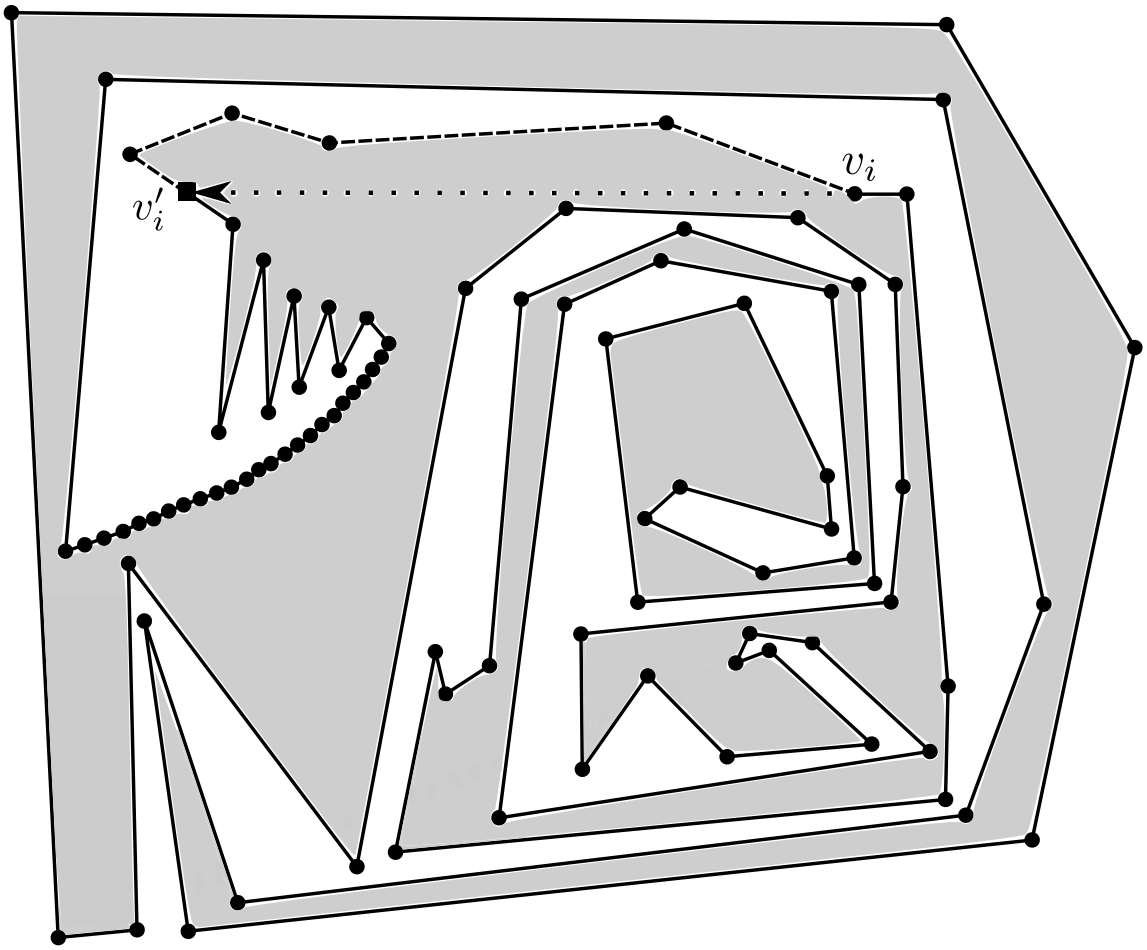

Figure 2.4: The function `getReflexMinFreeComp` chooses any reflex vertex v_i, and then finds the hitpoint of its clockwise ray v_i' by brute force. The chain $P[v_i', v_i]$ is $cw(v_i)$. This component is also known as P_0: the vertex v_i becomes w_0, and v_i' becomes w_0^*. The component P_0 is stored in `W` and then passed to `scanner`. The component P_0 is soild and D_0 is dashed.

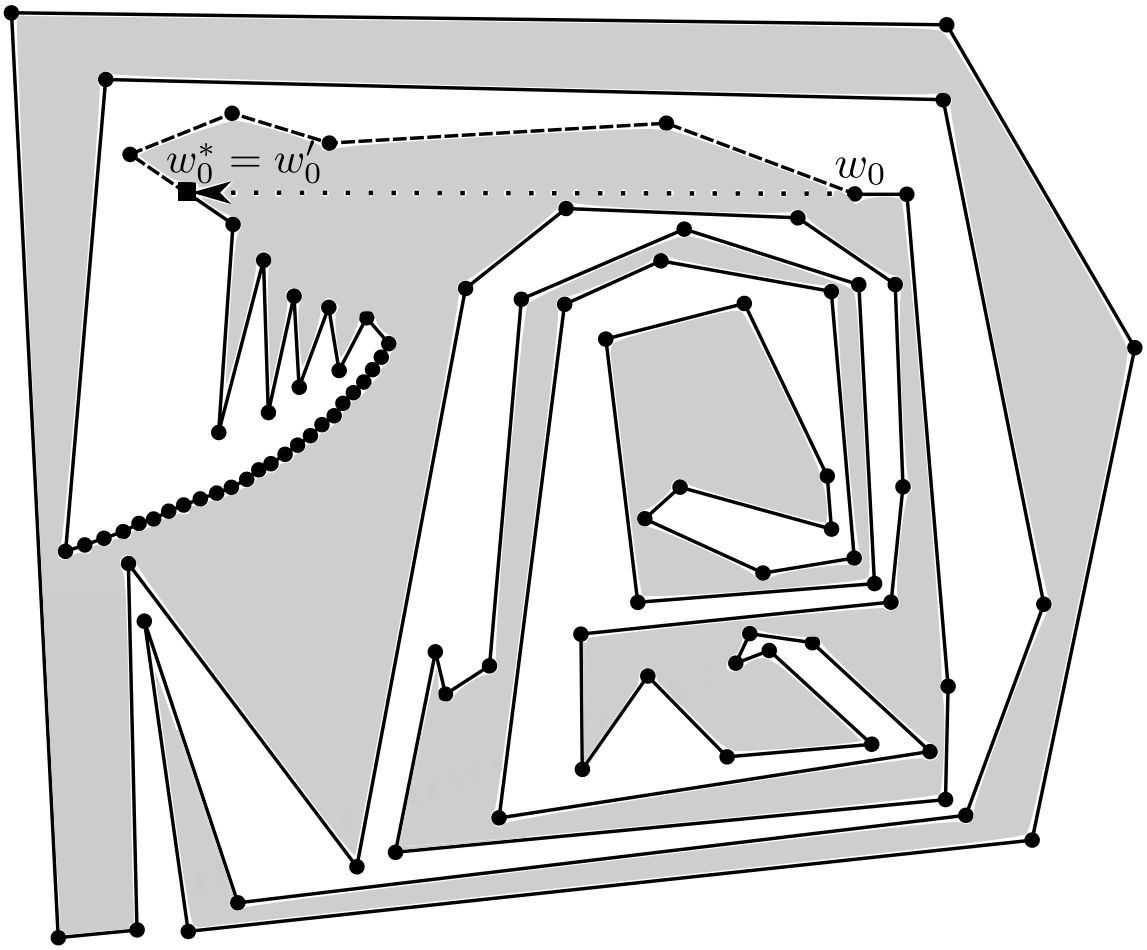

Figure 2.5: The clockwise and counterclockwise walks of `scanner` start from w_0 and w_0^*, respectively.

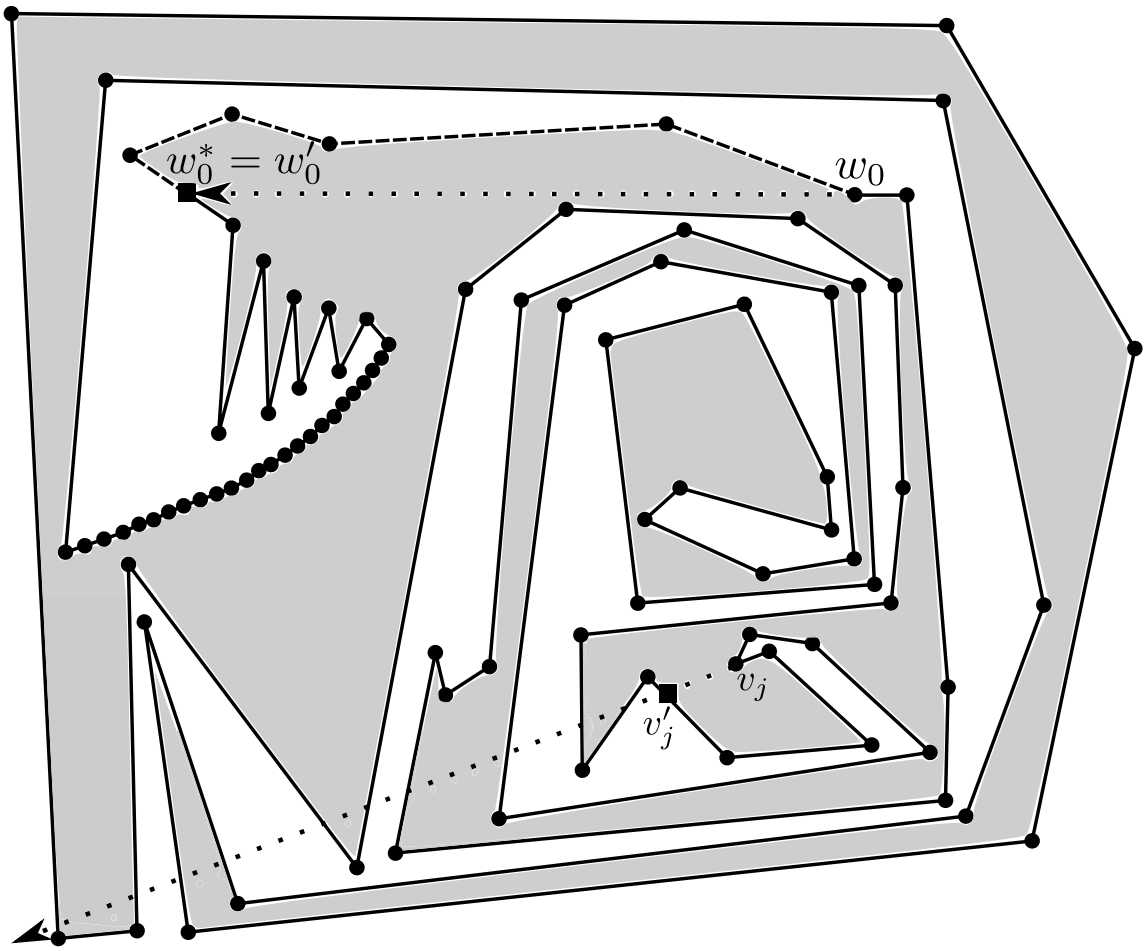

Figure 2.6: The clockwise and counterclockwise walks of `scanner` halt when the clockwise walk finds a cw-miss v_j. The vertex v_j is a cw-miss since it is the first time the clockwise walk from w_0 has encountered more reflex-minima vertices than convex maximas. Next `scanner` calls `cwMissChoose` which will either call `cwSegCut` or `cwRayCut`. In the next figure v_j becomes w_1 and we search for and then find w_1^*.

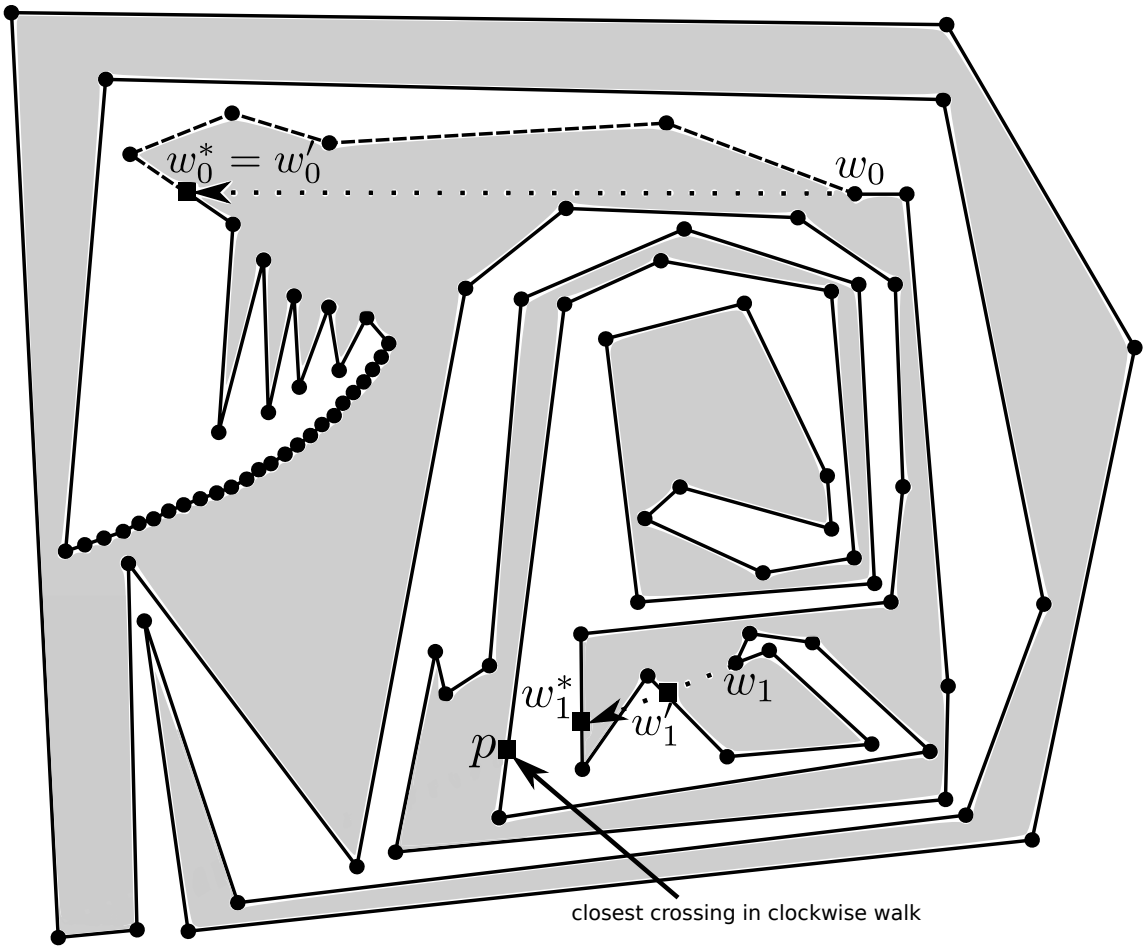

Figure 2.7: The function `cwMissChoose` walks $P[w_1, w_0]$ and finds the closest crossing between $\overrightarrow{cv}(w_1)$ and this chain, if it exists. Since there is such a crossing p, the function `cwMissChoose` calls `cwSegCut`. The function `cwSegCut` walks counterclockwise from w_0^* until it crosses the segment pw_1. This first crossing becomes w_1^*. We delete the chains $P[w_1, w_0]$ and $P[w_0^*, w_1^*]$.

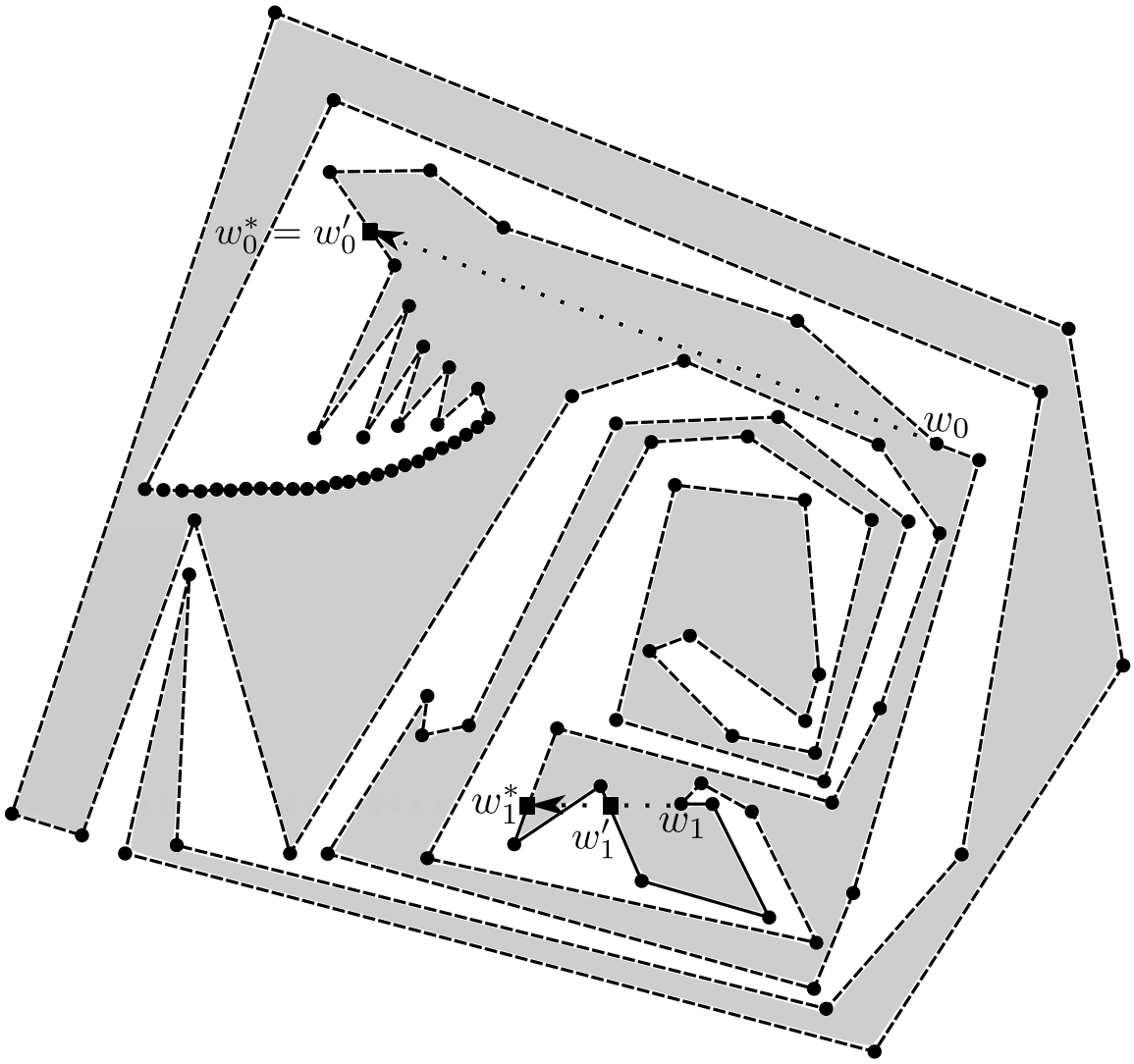

Figure 2.8: We now recurse on P_1. Again P_1 is solid and $D_1 = P[w_1, w_1^*]$ is dashed. The function `scanner` again begins its clockwise and counterclockwise walks from w_1 and w_1^*, respectively. The counterclockwise walk of `scanner` finds a crossing with $con(P_1)$ and calls `cwConnectorCut` since P_1 is a candidate cw-component. This function walks counterclockwise from w_1^* until finding a crossing that will become w_2^*. We delete the chain $P[w_1^*, w_2^*]$.

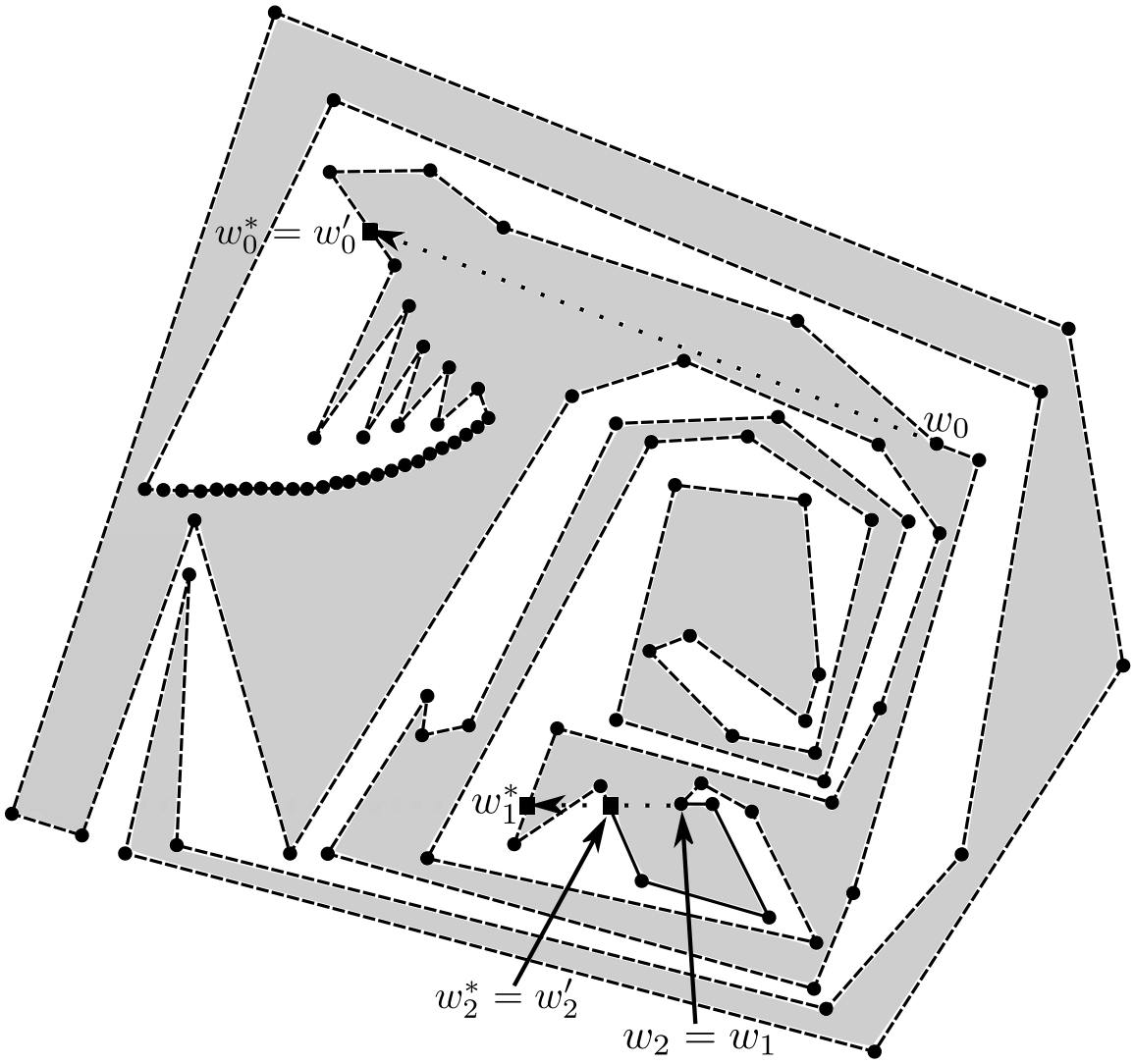

Figure 2.9: We now recurse on P_2. Again P_2 is solid and $D_2 = P[w_2, w_2^*]$ is dashed. The function `scanner` again begins its clockwise and counterclockwise walks from w_2 and w_2^*, respectively. This time both walks of `scanner` terminiate without event, thus P_2 has no reflex minima vertices and no crossings with $con(P_2)$, so it is a reflex-minima-free component. The algorithm outputs P_2.

2.3 The algorithm

We do not suggest a thorough reading of the algorithms at this point. It is best to read it in conjunction with the proofs.

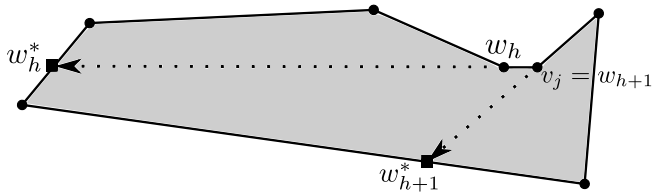

Figure 2.10: The vertex v_j is a cw-miss, but `scanner` will not select it since it is not technically a reflex minima.

(See Figure 2.10) Note that if P_h is a candidate cw-component, and the second clockwise neighbor of w_h is above $con(P_h)$, then the clockwise neighbor of w_h is a cw-miss even though it is not technically a minima vertex. We will consider this vertex to be a reflex minima and we will do the same for the symmetric counterclockwise case.

Algorithm 1 getReflexMinFreeComp(Chain V)

```
{find reflex vertex, if there is one}
Vertex x = V[0]
while  x is not reflex  do
      x = V.cwNeigh(x)
      if  x == V[0]  then
            return  "polygon is convex, contains no components"
      end if
end while

{find hitpoint of cw→(x) by brute force}
Point closest = null
for  each edge e in V  do
      if e crosses cw→(x) closer than closest then
            closest = crossing point with e
      end if
end for

{creates W, which stores cw(x)}
Chain W = V.setInitialCwComponent(x, closest)

scanner(W)
```

Algorithm 2 scanner(Chain W)

```
Integer cwExtremums = 0
Integer ccwExtremums = 0
Boolean cwWalkActive = true
Boolean ccwWalkActive = true
Vertex x = W.ccwEnd
Vertex y = W.cwEnd

while x ≠ W.cwEnd do

    {this is a clockwise step which examines only one vertex}
    if cwWalkActive then
        if W.cwEdge(x) crosses W.connector then
            if W stores a candidate cw-component then
                cwConnectorCut( W )
            else {W stores a candidate ccw-component}
                ccwConnectorCut( W )
            end if
            scanner( W )
            return
        else if x is above W.connector then
            cwWalkActive = false
        else if x is a convex maxima then
            cwExtremums++
        else if x is a reflex minima then
            cwExtremums−−
        end if
        if cwExtremums == −1 then
            cwMissChoose( W, x )
            scanner( W )
            return
        end if
        x = W.cwNeigh(x)
    end if

    {counterclockwise step is symmetric and is omitted}

end while
return W
```

Algorithm 3 cwConnectorCut (Chain W)

```
Integer crossingCounter = 0
Point closestCrossing = (∞, ∞)

{This walk checks each edge in W.}
for w ∈ W in ccw order from W.cwEnd do
    if W.ccwEdge(w) crosses W.connector then
        if crossing point is closer to W.ccwEnd than closestCrossing then
            closestCrossing = crossing point
        end if

        if crossing point is ext-left with respect to W.connector then
            crossingCounter++
        else
            crossingCounter−−
        end if

        if crossingCounter == 0 and
           crossing point == closestCrossing and
           crossing point is ext-left with respect to W.connector then
            W.delete(W.cwEnd, closestCrossing)
            return
        end if
    end if
end for
```

Algorithm 4 cwMissChoose (Chain W, Vertex cwMiss)

```
Point closestCrossing = (∞, ∞)
for each Vertex w in ccw order from cwMiss to W.cwNeigh(W.ccwEnd) do
    if W.cwEdge(w) crosses cw→(cwMiss) and w is closer to cwMiss than closestCross-
    ing then
        closestCrossing = crossing point
    end if
end for
if closestCrossing == (∞, ∞) then
    cwRayCut( W, cwMiss )
else
    cwSegCut( W, cwMiss, closestCrossing )
end if
```

Algorithm 5 cwRayCut (Chain W, Vertex cwMiss)

```
Integer crossingCounter = 0
Point closestCrossing = (∞, ∞)

{This walk checks each edge in W.}
for w ∈ W in ccw order from W.cwEnd do
    if W.ccwEdge(w) crosses W.connector then
        if W stores a candidate cw-component then
            cwConnectorCut( W )
        else {W stores a candidate ccw-component}
            ccwConnectorCut( W )
        end if
        return

    else if W.ccwEdge(w) crosses cw⃗(cwMiss) then
        if crossing point is closer to cwMiss than closestCrossing then
            closestCrossing = crossing point
        end if

        if crossing point is ext-left with respect to cw⃗(cwMiss) then
            crossingCounter++
        else
            crossingCounter−−
        end if

        if crossingCounter == 1 and
           crossing point == closestCrossing and
           crossing point is ext-left with respect to cw⃗(cwMiss) then
            W.delete(W.cwEnd, closestCrossing)
            W.delete(cwMiss, W.ccwEnd)
            return
        end if
    end if
end for
```

Algorithm 6 cwSegCut (Chain W, Vertex cwMiss, Point closestHit)

```
for w ∈ W in ccw order from W.cwEnd do
    if W.cwEdge(w) crosses W.connector then
        if W stores a candidate cw-component then
            cwConnectorCut( W )
        else {W stores a candidate ccw-component}
            ccwConnectorCut( W )
        end if
        return
    else if W.cwEdge(w) crosses segment cwMiss,closestHit then
        Point q = crossing point
        W.delete(W.cwEnd, q)
        W.delete(cwMiss, W.ccwEnd)
        return
    end if
end for
```

2.4 Preliminary lemmas

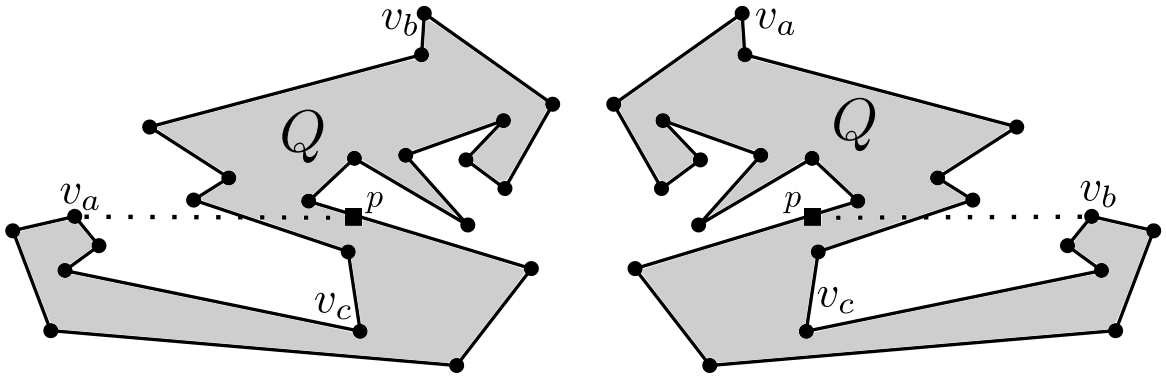

Figure 2.11: In both figures the chain $P[v_a, v_b]$ contains no reflex minima vertices, and only contains the convex maxima vertices v_a and v_b. However in the left figure, v_a is lower than v_b.

Lemma 1. *Let Z be a polygon with no edge parallel to the horizontal axis. Then, Z contains exactly one more convex maxima vertex than reflex minima vertex.*

Proof. We will use strong induction on the number of reflex minima vertices in Z.

Base Case:

Let the polygon Q has no reflex minima vertices. We will show that Q has exactly one convex maxima vertex. We know that the highest vertex of Q is a convex maxima. Thus, we need to show that Q cannot have more than one convex maxima vertex without having a reflex minima. Assume this for the purposes of contradiction.

(See Figure 2.11) Choose two convex maxima vertices $v_a, v_b \in Q$, such that $Q[v_a, v_b]$ does not contain another convex maxima vertex. There are two possibilities. Either v_a is below v_b (See the left hand side of Figure 2.11), or v_b is below v_a (See the right hand side of Figure 2.11). Since both cases are symmetric, without loss of generality, we will consider the case where v_a is lower.

We will shoot a horizontal ray from v_a to the right. Let p be the hitpoint of this ray with $Q[v_a, v_b]$. The chain $Q[v_a, p]$ and $v_a p$ forms a polygon R.

Since v_a is a local maxima, it's clockwise neighbor is in the interior of R. Let $v_c \in P[v_b, v_a]$ be in the interior of R and be lower than any other such

vertex. Clearly, v_c is a reflex minima vertex. Thus Q contains a reflex minima, a contradiction. Therefore, a polygon cannot have more than one convex maxima vertex without containing a reflex minima.

Inductive hypothesis:

There exists a constant k such that every polygon with $j \leq k$ reflex minima vertices contains exactly $j + 1$ convex maxima vertices.

Inductive step:

(See Figure 2.12) Let the polygon P have $k + 1$ reflex minima vertices. We will show that P has exactly $k + 2$ convex maxima vertices.

Choose a reflex minima vertex $v_q \in P$ and shoot a ray straight down. Let the hitpoint be q. Note that two vertices may be directly above each other, but q cannot be a reflex minima or convex maxima vertex. The chord qv_q partitions P into two sub-polygons: P^1, which is $P[v_q, q]$ and $v_q q$, and P^2 which is $P[q, v_q]$ and $v_q q$. The polygons P, P^1, and P^2 respectively contain $r = k + 1$, r_1, and r_2 reflex minima vertices and c, c_1, and c_2 convex maxima vertices.

By the inductive hypothesis, we know that $c_1 = r_1 + 1$ and $c_2 = r_2 + 1$. The vertex v_q is a reflex minima vertex in P, but not in P^1 or P^2. Thus $r = r_1 + r_2 + 1$. Also note that v_q is not a convex maxima in P, P^1, or P^2. Thus, $c = c_1 + c_2$. Therefore, $c = c_1 + c_2 = r_1 + r_2 + 2 = r + 1$, so $c = r + 1$. Thus, Q has one more convex maxima than reflex minima vertex.

□

We will show that each D_h is non-interfering by induction on h. This property serves as the central invariant for this chapter. Lemma 2, which follows, is the base case for Lemma 13, the induction proof that shows that this invariant holds for each each time P_h is cut to yield P_{h+1}. If P_h is not cut to yield P_{h+1}, then Lemma 15 shows that P_h is a reflex-minima-free component.

Lemma 2. *If P_h is a component, then D_h is non-interfering.*

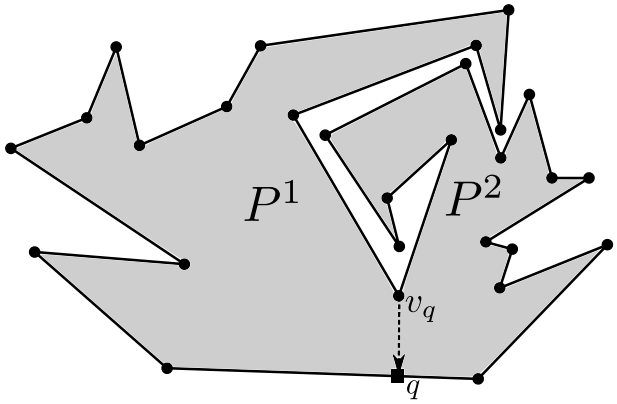

Figure 2.12: We partition P into P^1 and P^2 by shooting a ray straight down from v_q.

Proof. The chain D_h has no crossings with $con(P_h)$, so it is trivially balanced and not ext-right heavy with respect to $con(P_h)$. Also, since $con(P_h)$ is a chord in P, we know that in the neighborhoods of both w_h and w_h^*, the segment $con(P_h)$ is in the interior of P. Thus $con(P_h)$ is a pseudochord. Therefore, D_h is non-interfering. □

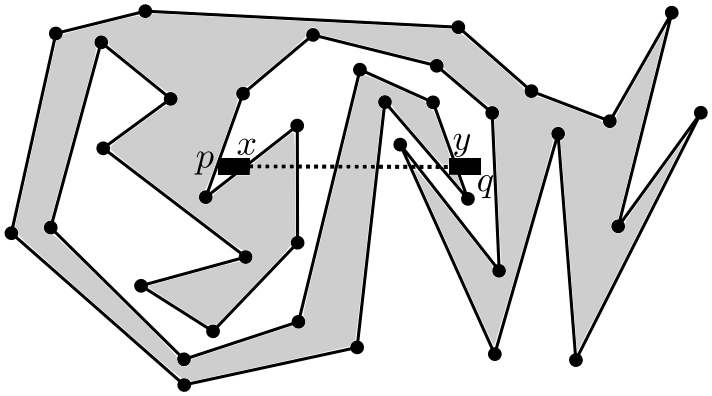

Figure 2.13: The segment xy is a pseudochord in P, not a chord, because in the neighborhoods of both x and y, the segment is in the interior of P.

Lemma 3. *Let xy be a pseudochord in P. The chain $P[x, y]$ is balanced with respect to xy if and only if $P[y, x]$ is as well.*

Proof. (See Figure 2.13) We will consider P to be in standard position about xy so that x is to the left of y and xy is horizontal. Let p, q be in the line that contains xy, where p is ϵ distance to the left of x and q is ϵ distance to the right of y. If ϵ is small enough, then

P cannot cross px and yq. Note that x and y are crossings with respect to pq. Since xy is a pseudochord, we know that with respect to pq, the crossings x and y cannot both be ext-right or ext-left, they must be different.

First we will show that if $P[x, y]$ is balanced with respect to xy, then $P[y, x]$ is as well. Since x and y are not both ext-left or both ext-right with respect to pq, we know that $P[x, y]$ is also balanced with respect to pq.

If we walk along pq from p to q, by the Jordan Curve theorem the crossings we visit will alternate between ext-left and ext-right crossings. Therefore, since x and y are not both ext-left or ext-right crossings, we know that P is balanced with respect to pq.

Since P and $P[x, y]$ are balanced with respect to pq, by the Pigeon Hole principle $P[y, x]$ is balanced with respect to pq as well. Note that if we analyze the crossings between $P[y, x]$ and xy instead of pq, the points x and y are no longer crossings. However, since x and y are not both ext-left or ext-right crossings, we know that if $P[y, x]$ is balanced with respect to pq, then it is also balanced with respect to xy.

The proof in the other direction is an identical argument. We will show that if $P[y, x]$ is balanced with respect to xy, then $P[x, y]$ is as well. Since x and y are not both ext-left or ext-right crossings, we know that $P[y, x]$ is balanced with respect to pq. Since x and y are not both ext-left or ext-right crossings, and since by the Jordan Curve theorem, crossings alternate as we walk from p to q, we know P is balanced with respect to pq. Since P and $P[y, x]$ are balanced with respect to pq, by the Pigeon Hole principle, $P[x, y]$ is balanced with respect to pq. Finally since x and y are not both ext-left or ext-right crossings, we know that $P[x, y]$ is balanced with respect to xy as well. Thus, $P[x, y]$ is balanced with respect to xy if and only if $P[y, x]$ is. □

Lemma 4, which follows, proves a fundamental property of our non-interfering invariant that will be used by several other lemmas.

Lemma 4. *If D_h is non-interfering, then $w'_h \in P_h$.*

Proof. (See Figure 2.14) Assume for the purposes of contradiction that $w'_h \in D_h$. Let the vertex that is immediately clockwise of w'_h be v_p. We know v_p is above w'_h. Let v_q be the counterclockwise neighbor of w_h, which is also above w_h. Let $p \in w'_h v_p$ be ϵ distance from w'_h and let $q \in w_h v_q$ be ϵ distance from w_h. If ϵ is small enough, we know that pq must be a chord in P because $w'_h w_h$ is a chord in P.

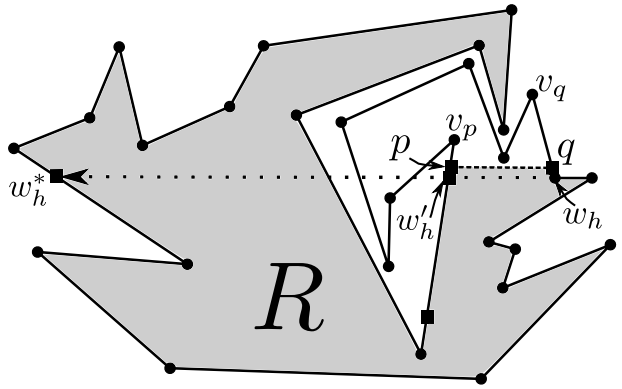

Figure 2.14: The shaded region is R. This is an example of when $w_h' \in D_h$. Observe that D_h is ext-right heavy with respect to $con(P_h)$ and thus it is interfering.

Let R be the polygon bound by $P[p, q]$ and qp. Since D_h is non-interfering, it is balanced with respect to $con(P_h)$, which is the segment $w_h w_h^*$. Since D_h is non-interfering, $con(P_h)$ is a pseudochord. Thus, by Lemma 3, P_h is also balanced with respect to $con(P_h)$. Since $R[w_h^*, w_h]$ is the same as P_h, we know that $R[w_h^*, w_h]$ is also balanced with respect to $con(P_h)$. Thus, by Lemma 3, since $con(P_h)$ is a pseudochord, $R[w_h, w_h^*]$ is also balanced with respect to $con(P_h)$.

Note that since pq is a chord, the only crossing with $con(P_h)$ in $R[w_h, w_h']$ is w_h'. Therefore, the chains $R[w_h, w_h^*]$, $R[w_h', w_h^*]$, and $P[w_h', w_h^*]$ have the same crossings with $con(P_h)$. Since the former is balanced with respect to $con(P_h)$, then so is the latter.

Consider traversing $P[w_h', w_h^*]$ clockwise from w_h^*, the crossing w_h' is the last we encounter. Since w_h' is a ext-left-crossing and since $P[w_h', w_h^*]$ is balanced with respect to $con(P_h)$, we know that the previous crossing that we encountered must be ext-right heavy with respect to $con(P_h)$. Thus, D_h is ext-right heavy and therefore interfering, a contradiction. Thus, $w_h' \in P_h$. □

The following two lemmas, 2.16 and 2.17, show that both the clockwise and counterclockwise walks of `scanner` cannot both stop at the same time. Lemma 2.16 is a helper for Lemma 2.17. Together, these lemmas show that if the counterclockwise walk halts and sets `ccwWalkActive` to false, that the clockwise walk will find a cw-miss, and therefore will not halt. The symmetric also holds, that if the clockwise walk halts, the counterclockwise walk of `scanner` will find a ccw-miss and not halt.

Lemma 5a. *If D_h is non-interfering, and the clockwise walk of `scanner` reaches the vertex*

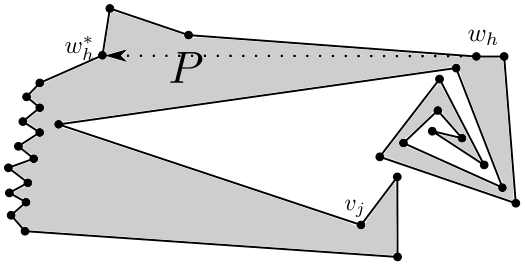

Figure 2.15: This is the polygon P, that R in Figure 2.16 is based on. The function `scanner` will find the cw-miss v_j, where v_j is the lowest vertex in $R[v_j, w_h]$.

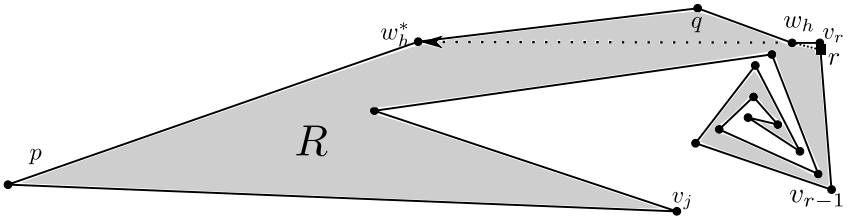

Figure 2.16: The chain $R[v_j, w_h]$ comes from an underlying polygon P, depicted in the above Figure 2.15. The polygon R is constructed around $P[v_j, w_h]$. Observe that the vertex v_j is a reflex minima vertex in P, but not in R.

v_j, where v_j is reflex, and is the lowest vertex in the walk, then `scanner` will call `cwMissChoose` upon reaching v_j, if not before.

Proof. The if-statement that encloses the call of `cwMissChoose` requires that v_j be a reflex minima vertex and that `cwExtremums == -1` when `scanner` reaches v_j. We already know that the former holds, so we will prove the latter.

We will show that $P[v_j, w_h]$ has exactly one more reflex minima than convex maxima vertex. From this it follows that `cwExtremums == -1` upon reaching v_j. Towards this goal, we will construct a polygon R that includes $P[v_j, w_h]$ and then apply Lemma 1.

(See Figure 2.16) Since `scanner` reached v_j, we know that no point of $P[v_j, w_h]$ is above $con(P_h)$. Using this fact, we will construct R in three steps:

1. Let q be a point in the half-plane above the connector. Since $P[v_j, w_h]$ is entirely below $con(P_h)$, we know that this chain will not cross w_h^*q or w_hq.

2. Let p be below $con(P_h)$, above v_j, and far enough to the left of any vertex in $P[v_j, w_h]$ that pv_j and pw_h^* do not cross $P[v_j, w_h]$.

3. Let v_r be the clockwise neighbor of w_h, and let $r \in v_r v_{r-1}$ be ϵ distance from v_r. To enable us to apply Lemma 1, we replace v_r with r.

In counterclockwise order, the polygon R will be composed of $w_h q$, $q w_h^*$, $w_h^* p$, $p v_j$, $P[v_j, r]$, and then $r w_h$. Note that in $R[w_h, v_j]$, the only vertex that is a convex maxima or reflex minima vertex in R is the convex maxima q. Note that the vertex v_j is a reflex minima in P but not in R.

Since R has no horizontal edges, we know by Lemma 1 that R has one more convex maxima than reflex minima vertex. Let the polygon $Q \subset P$ be the part of R that is not above $con(P_h)$. Since q was a convex maxima in R, we know that Q has exactly as many reflex minima vertices as convex maxima vertices.

Since the only vertices in Q that are reflex minimas or convex maximas are in $P[v_j, w_h]$, we know that this chain has exactly as many reflex minimas as convex maximas. Thus, immediately before reaching v_j, we know that cwExtremums == 0. Therefore, since v_j is a reflex minima in P, we know that when `scanner` reaches v_j, that cwExtremums == -1. Thus, `scanner` will call `cwMissChoose` upon reaching the reflex minima vertex v_j, if it did not already call it for another vertex in $P[w_h, v_j]$. □

Lemma 5b. *If D_h is non-interfering, and the counterclockwise walk of `scanner` reaches the vertex v_j, where v_j is reflex, and is the lowest vertex in the walk, then `scanner` will call `ccwMissChoose` upon reaching v_j, if not before.*

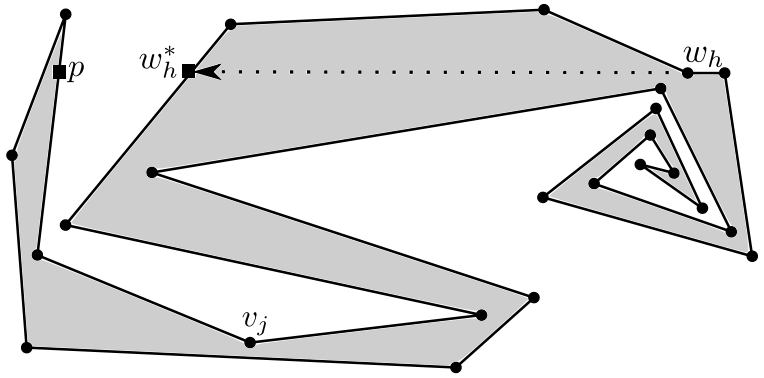

Figure 2.17: The counterclockwise walk of `scanner` walks above the connector at the point p. We will show a contradiction, since the counterclockwise walk of `scanner` would have found the ccw-miss v_j before reaching p.

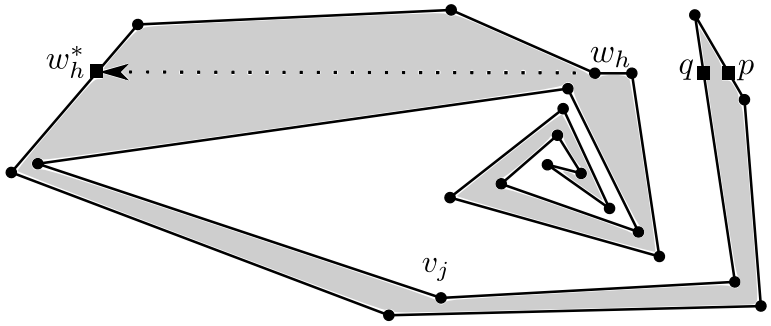

Figure 2.18: The counterclockwise walk of `scanner` walks above the connector at the point p. Then `scanner` sets `ccwWalkActive` to false. We will show that the clockwise walk of `scanner` will always find a cw-miss before crossing w_hp, in this case v_j.

Recall that if the clockwise walk of `scanner` walks above the line containing the connector, without crossing the connector, then `cwWalkActive` will be set to false. The next lemma shows that if one walk is halted in this way, the other walk will cut P_h.

Lemma 6a. *If D_h is non-interfering, and in `scanner` the variable `ccwWalkActive` is false, then the clockwise walk of `scanner` will find a cw-miss or cross $con(P_h)$.*

Proof. (See Figures 2.17, and 2.18) Without loss of generality, let P_h be a candidate cw-component. Since `ccwWalkActive` is false, we know that the counterclockwise walk of `scanner` crosses the line containing $con(P_h)$ at the point p, where $p \notin con(P_h)$. There are two cases, either p is to the left of w_h^* or it is to the right of w_h.

p is to the left of w_h^*:

(See Figure 2.17) There exists a lowest vertex, $v_j \in P[w_h^*, p]$, which is a reflex minima. Thus, by Lemma 5b, `scanner` would have recognized v_j as a ccw-miss and called `ccwMissChoose`. Thus, we know p is to the right of w_h.

p is to the right of w_h:

(See Figure 2.18) We know that $P[w_h^*, p]$ and pw_h^* form a polygon Q. Thus, if we walk clockwise from w_h, we must either cross $con(P_h)$, in which case we call `cwConnectorCut`, or cross w_hp at the point q. Assume the latter. The chain $P[q, w_h]$ has a minimum vertex v_j which is a reflex minima. Thus, by Lemma 5a, `scanner` would have recognized v_j as a cw-miss and called `cwMissChoose`.

Therefore, if `ccwWalkActive == false`, the clockwise walk of `scanner` will eventually find a cw-miss or cross $con(P_h)$.

□

Lemma 6b. *If D_h is non-interfering, and in `scanner`, the variable `cwWalkActive` is false, then the counterclockwise walk of `scanner` will find a ccw-miss or cross $con(P_h)$.*

2.5 If D_h is non-interfering, then D_{h+1} is also non-interfering

Lemma 2 will serve as the basis for the induction proof that will show that each D_h is non-interfering. This section presents several lemmas which together form the inductive step for the proof showing that this invariant is maintained.

In the first subsection we will show that if D_h is non-interfering, then when called, `cwConnectorCut` and `ccwConnectorCut` terminate and cut P_h in such a way that P_{h+1} is also non-interfering. In the second subsection, we will show the same for `cwSegCut`, `ccwSegCut`, `cwRayCut`, and `ccwRayCut`.

2.5.1 Cutting crossings with $con(P_h)$ maintains non-interfering invariant

(See Figure 2.19) If `scanner`, or one of the other cutters, discovers a crossing with $con(P_h)$, it calls `cwConnectorCut` if P_h is a candidate cw-component and `ccwConnectorCut` if it is a candidate ccw-component. We give details on `cwConnectorCut` but not the symmetric `ccwConnectorCut`.

The function `cwConnectorCut` walks counterclockwise from w_h^* until it finds a crossing x with $con(P_h)$. Lemma 7a shows that it will indeed find a crossing and that x has the following properties:

- x is the closest crossing to w_h in the chain the walk visited, $P[w_h^*, x]$,
- x is an ext-left-crossing, and
- including x, the walk has visited as many ext-left-crossings as ext-right-crossings.

Upon finding x, `cwConnectorCut` deletes the chain it walked, $P[w_h^*, x]$. What remains of P_h will become P_{h+1}, the crossing x will become w_{h+1}^*, and w_h will also be w_{h+1}. Finally, Lemma 8a shows that the candidate cw-component, P_{h+1}, produced by `cwConnectorCut`, is non-interfering.

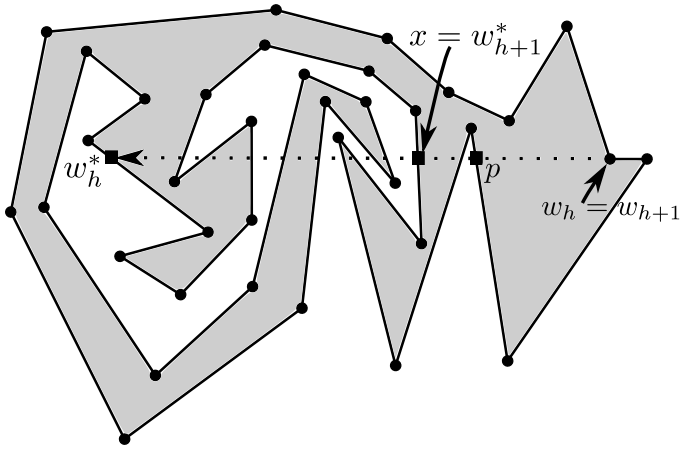

Figure 2.19: The clockwise step of `scanner` will find the crossing p with $con(P_h)$. Since P_h is a candidate cw-component, it calls `cwConnectorCut`, which finds x and deletes $P[w^*_h, x]$. The vertex x becomes w^*_{h+1} and w_{h+1} will be w_h.

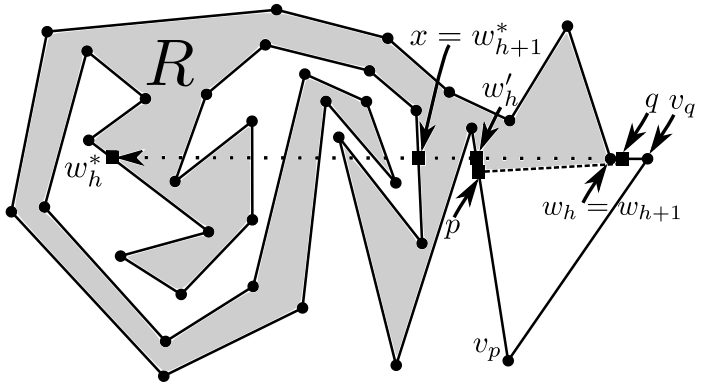

Figure 2.20: The interior of R is shaded. Note that `cwConnectorCut` would select x and not w'_h. However, w'_h is a satisfying crossing.

Lemma 7a. *Let D_h be non-interfering, and let P_h be a candidate cw-component. If `cwConnectorCut` is called on P_h, then it will find a crossing $x \in P_h$ with $con(P_h)$, such that: (1) x is an ext-left-crossing, (2) $P[w^*_h, x]$ is balanced with respect to $con(P_h)$, and (3) x is the closest crossing to w_h in $P[w^*_h, x]$.*

Proof. (See Figure 2.19 in which $p = w'_h$) If `cwConnectorCut` cuts P_h at a crossing y, we know by the conditions of the if-statement that encloses `W.delete`, that the following holds for y: 1) `crossingCounter == 0`, 2) `closestCrossing ==` y, and 3) y is an ext-left-crossing with respect to `W.connector`. These three conditions are equivalent to the three conditions for x.

Now we only need to show that `cwConnectorCut` will always find a crossing at which to cut. We will show that w'_h is such a crossing. By Lemma 4, we know that $w'_h \in P_h$. Thus, since `cwConnectorCut` traverses P_h, it will reach w'_h, unless it finds another satisfying crossing first. Thus, we only need to show that w'_h satisfies the three conditions for x.

Note that by definition w'_h is the closest crossing to w_h and by the Jordan Curve Theorem, it must be an ext-left-crossing. Thus, we only need to show that when w'_h is reached that `crossingCounter == 0`, or in other words: $P[w^*_h, w'_h]$ is balanced with respect to $con(P_h)$. This argument will be very similar to the argument in Lemma 4, however in that lemma, w'_h was assumed to be in D_h instead of P_h.

(See Figure 2.20) Let the vertex that is immediately counterclockwise of w'_h be v_p. We know v_p is below w'_h. Similarly, let v_q be the vertex that is clockwise of w_h. We know v_q is horizontal and right of w_h. Let $p \in w'_h v_p$ be ϵ distance from w'_h and let $q \in w_h v_q$ be ϵ distance from w_h. If ϵ is small enough, we know that pq must be a chord in P because $w'_h w_h$ is a chord in P.

Let R be the polygon bound by $P[q, p]$ and pq. Since D_h is non-interfering, it is balanced with respect to $con(P_h)$. Since $R[w_h, w^*_h] = D_h$, it is also balanced with respect to $con(P_h)$. Since D_h is non-interfering, $con(P_h)$ is a pseudochord. Thus, by Lemma 3, we know that $R[w^*_h, w_h]$ is also balanced with respect to $con(P_h)$. Since $R[w^*_h, w_h]$ and $P[w^*_h, w'_h]$ have the same crossings with $con(P_h)$, we know that the latter chain is also balanced with respect to $con(P_h)$. Thus, if the walk reaches w'_h, we know that `crossingCounter = targetCrossingVal = 0`.

We have shown that the walk will reach w'_h if it does not find another satisfying crossing first, and that the three properties in the proof statement will be satisfied.

□

Lemma 7b. *Let D_h be non-interfering, and let P_h be a candidate ccw-component. If `ccwConnectorCut` is called on P_h, then it will find a crossing $x \in P_h$ with $con(P_h)$, such that: (1) x is an ext-right-crossing, (2) $P[x, w^{**}_h]$ is balanced with respect to $con(P_h)$, and (3) x is the closest crossing to w_h in $P[x, w^{**}_h]$.*

Now we will show that if D_h is non-interfering, then after `cwConnectorCut` cuts P_h, the chain D_{h+1} will also be non-interfering. Recall that `ccwConnectorCut` keeps track of the balance of ext-left and ext-right crossings it has encountered in its walk with the variable `crossingCounter`.

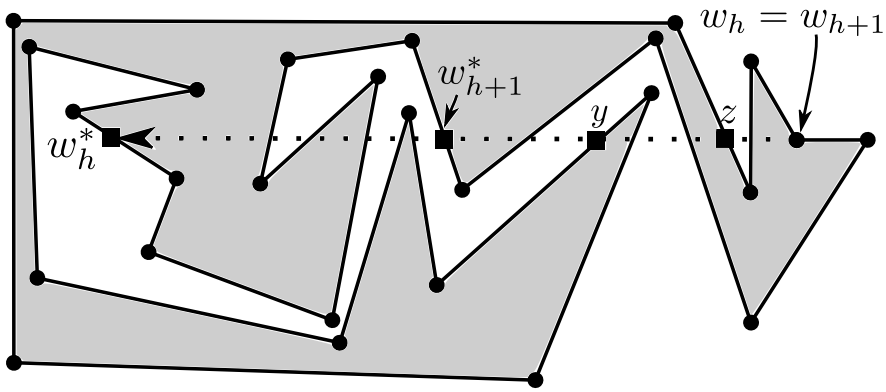

Figure 2.21: Observe that D_h is ext-right heavy with respect to $con(P_h)$ and thus interfering. Also observe that $P[z, w^*_h]$ is balanced, so it does not contain more ext-left crossings than ext-rights with respect to $w^*_h w^*_{h+1}$.

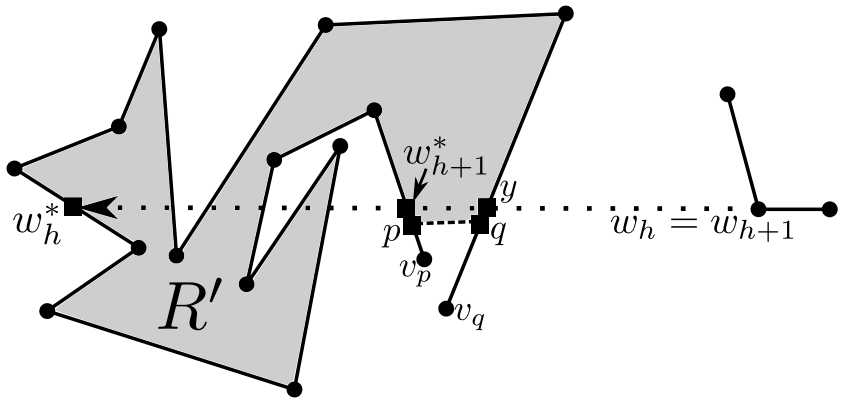

Figure 2.22: This figure shows that if y is an ext-right-crossing, then D_h is ext-right heavy with respect to $con(P_h)$, and thus D_h is interfering.

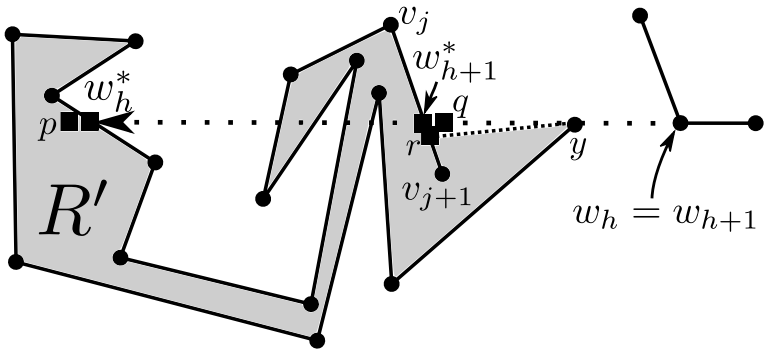

Figure 2.23: We will analyze the crossings of $P[y, w^*_h] = R'[w^*_h, y]$ with respect to both $w^*_h w^*_{h+1}$ and pq.

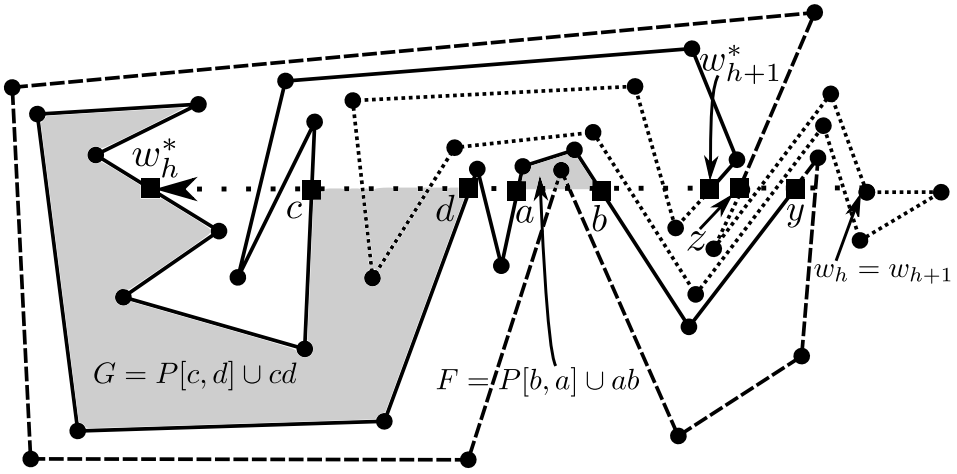

Figure 2.24: The polygons F and G are shaded. The chain $P[y, w^*_{h+1}] \subset R$ is drawn solid, the chain $P[z, y]$ is dashed, and the rest of P is dotted. The ext-left-crossing y is the first crossing with $con(P_{h+1})$ found in a clockwise walk of D_h from w^*_h, and z is the first crossing where D_h is ext-right heavy with respect to $con(P_{h+1})$. Note that $a, b, c, d \in R$, and that ab and cd do not cross R, however cd is a chord in R and ab is not. Finally, observe that $P[z, y]$ is balanced with respect to ab and cd, as well as $w^*_h w^*_{h+1}$.

Lemma 8a. *Let D_h be non-interfering. If* `cwConnectorCut` *is called on P_h, then when it terminates, D_{h+1} will be non-interfering as well.*

Proof. (See Figure 2.19) Without loss of generality, let P_h be a candidate cw-component. Recall that D_{h+1} is composed of the chains D_h and $P[w^*_h, w^*_{h+1}]$. Also recall that D_{h+1} is non-interfering if $con(P_{h+1}) = w^*_{h+1} w_h$ is a pseudochord, and D_{h+1} is balanced and not ext-right heavy with respect to $con(P_{h+1})$.

By Lemma 7a, we know that w^*_{h+1} is an ext-left crossing with $con(P_h)$, that $P[w^*_h, w^*_{h+1}]$ does not cross $con(P_{h+1})$, that this chain is balanced with respect to $con(P_h)$, and that `cwConnectorCut` will always find a crossing in P_h. This implies that $P[w^*_h, w^*_{h+1}]$ has one more ext-right-crossing than ext-left with respect to $w^*_h w^*_{h+1}$.

$con(P_{h+1})$ **is a pseudochord:**

> Since D_h is non-interfering, we know that in the neighborhood of w_h, the segment $con(P_h)$ is in the interior of P. Since $w_h = w_{h+1}$, we know that in the neighborhood of w_{h+1}, the segment $con(P_{h+1})$ will also be in the interior of P. Also, since w^*_{h+1} is an ext-left crossing with $con(P_h)$, we know that in

the neighborhood of w^*_{h+1}, the segment $con(P_{h+1})$ is in the interior of P. Thus, $con(P_{h+1})$ is a pseudochord.

D_{h+1} is balanced with respect to $con(P_{h+1})$:

Because of these three facts: 1) P_h is balanced with respect to $con(P_h)$, 2) $P[w^*_h, w^*_{h+1}]$ has one more ext-right-crossing than ext-left-crossings with respect to $w^*_h w^*_{h+1}$, and 3) w^*_{h+1} is an ext-left crossing, we know that $P_{h+1} = P[w^*_{h+1}, w_h]$ is balanced with respect to $con(P_{h+1})$.

Since $con(P_{h+1})$ is a pseudochord, by Lemma 3, we know that because P_{h+1} is balanced with respect to $con(P_{h+1})$, and $con(P_{h+1})$ is a pseudochord, we know that D_{h+1} is also balanced with respect to $con(P_{h+1})$.

Now we only need to show that D_{h+1} is not ext-right heavy with respect to $con(P_{h+1})$. We know that D_h and D_{h+1} have the same crossings with $con(P_{h+1})$ and in the same order. Therefore, showing that D_{h+1} is non-interfering with respect to $con(P_{h+1})$ is the same as showing that D_h is. Since we know that D_h is non-interfering with respect to $con(P_h)$, it will be easier to use D_h, instead of D_{h+1}.

(See Figure 2.21) For the purposes of contradiction, let D_h be ext-right heavy with respect to $con(P_{h+1})$. In a clockwise walk of D_h from w^*_h, let y be the first crossing with $con(P_{h+1})$, and let z be the first time in the walk that $P[z, w^*_h]$ is ext-right heavy with respect to $con(P_{h+1})$. Since D_h is non-interfering, we know $P[z, w^*_h]$ is not ext-right heavy with respect to $con(P_h)$. Thus, $P[z, w^*_h]$ must contain more ext-left crossings than ext-rights with respect to $w^*_h w^*_{h+1}$.

We will use y and z to partition D_h into three subchains. We will first show that y is an ext-left crossing with respect to $con(P_{h+1})$. Then we will show that $P[y, w^*_h]$ and $P[z, y]$ are both balanced with respect to $w^*_h w^*_{h+1}$. This will be a contradiction since the combined chain, $P[z, w^*_h]$, is balanced and does not contain more ext-left crossings than ext-rights, with respect to $w^*_h w^*_{h+1}$.

y is an ext-left-crossing with respect to $con(P_{h+1})$:

(See Figure 2.22) For the purposes of contradiction, assume that y is an ext-right-crossing with $con(P_{h+1})$. We know that $P[w^*_h, w^*_{h+1}] \subset P_h$ does not cross $con(P_{h+1})$ and that $P[y, w^*_h] \subset D_h$ does not cross $w^*_{h+1} y$. Thus, the union of

these chains, $P[y, w^*_{h+1}]$, does not cross $w^*_{h+1}y$. Therefore this chain and segment together form a polygon R.

Let the vertex v_p be the counterclockwise neighbor of w^*_{h+1}, and let v_q be the clockwise neighbor of y. We know v_p and v_q are below $con(P_h)$. Let $p \in w^*_{h+1}v_p$ be ϵ distance from w^*_{h+1}, and let $q \in yv_q$ be ϵ distance from y. If ϵ is small enough, we know that $P[q, p]$ will not cross pq because $P[y, w^*_{h+1}]$ does not cross $w^*_{h+1}y$. Let $P[q, p]$ and pq form the polygon R'.

Since $P[w^*_h, w^*_{h+1}]$ has one more ext-right-crossing than ext-left with respect to $w^*_h w^*_{h+1}$, and since w^*_{h+1} is an ext-left-crossing, we know that $R'[w^*_h, y]$ is balanced with respect to $w^*_h y$.

We want to use Lemma 3 with respect to $w^*_h y$, so we must first show that it is a pseudochord. Since $con(P_h)$ is a pseudochord, and y is an ext-right-crossing, we know that in the neighborhood of w^*_h and y, the segment $w^*_h y$ is in the interior of P. Thus, $w^*_h y$ is a pseudochord.

Therefore, by Lemma 3, the chain $R'[y, w^*_h]$ is balanced with respect to this segment as well. Since $R'[y, w^*_h] = P[y, w^*_h]$ is balanced with respect to $w^*_h y$ in R', we know $P[y, w^*_h]$ is also balanced with respect to this segment in P.

If we analyze the crossings with $P[y, w^*_h]$ and $con(P_h)$, rather than $w^*_h y$, then we now include the ext-right-crossing y. Thus $P[y, w^*_h]$ is ext-right heavy with respect to $con(P_h)$. Therefore D_h is ext-right heavy with respect to $con(P_h)$. This means that D_h is interfering with $con(P_h)$, a contradiction. We conclude that y is an ext-left-crossing.

$P[y, w^*_h] \subset D_h$ is balanced with respect to $w^*_h w^*_{h+1}$:

(See Figure 2.23) Let w^*_{h+1} be in the edge $v_j v_{j+1}$ of P. We know that v_{j+1} is below $con(P_h)$. Let $r \in w^*_{h+1}v_{j+1}$ be ϵ distance from w^*_{h+1}. We know that $P[y, w^*_{h+1}]$ does not cross yw^*_{h+1}, so if ϵ is small enough, $P[y, r]$ will not cross yr either. Let $P[y, r]$ and yr form the polygon R'.

Note that what was an ext-left-crossing in P is an ext-right-crossing in R', and vice versa. Therefore, since $P[w^*_h, w^*_{h+1}]$ has one more ext-right than ext-left crossing with respect to $w^*_h w^*_{h+1}$ in P, we know that $R'[w^*_{h+1}, w^*_h] = P[w^*_h, w^*_{h+1}]$ has one more ext-left than ext-right crossing with respect to $w^*_h w^*_{h+1}$.

Let p and q be in the line through $con(P_h)$, where p is ϵ distance to the left of w_h^*, and q is ϵ to the right of w_{h+1}^*. If ϵ is small enough, P will not cross pw_h^* or w_{h+1}^*q. Note that w_h^* and w_{h+1}^* are both ext-right-crossings with respect to pq. Thus, $R'[w_{h+1}^*, w_h^*]$ has one more ext-right than ext-left crossing with respect to pq.

By the Jordan Curve theorem, because R' is a polygon, crossings along pq must alternate as we walk along pq from p to q. Thus, since the leftmost and rightmost crossings are ext-right, we know that R' also has one more ext-right than ext-left crossing with respect to pq.

Since R' and $R'[w_{h+1}^*, w_h^*]$ both have one more ext-right than ext-left crossing with respect to pq, and since the crossings along pq alternate, we know that the chain $R' \setminus R'[w_{h+1}^*, w_h^*]$ (which does not include the points w_{h+1}^* or w_h^*) must be balanced with respect to pq.

Note that since the chain $R' \setminus R'[w_{h+1}^*, w_h^*]$ does not include the crossings w_{h+1}^* and w_h^*, we know that this chain has the same crossings with pq as it does with $w_h^*w_{h+1}^*$. Thus, this chain is also balanced with respect to $w_h^*w_{h+1}^*$.

The chains $R' \setminus R'[w_{h+1}^*, w_h^*]$ and $R'[w_h^*, y] = P[y, w_h^*]$ also share the same crossings with $w_h^*w_{h+1}^*$. Thus, $P[y, w_h^*]$ is balanced with respect to $w_h^*w_{h+1}^*$.

$P[z, y]$ is balanced with respect to $w_h^*w_{h+1}^*$

(See Figure 2.24) Let R (not R') partition $w_h^*w_{h+1}^*$ into sub-segments. Some of these sub-segments are chords in R and the rest are entirely not in R.

First, let ab be a sub-segment of $w_h^*w_{h+1}^*$ that is entirely outside of R. Combining $R[a, b]$ with ab and $R[b, a]$ with ab, yields one finite polygon R and one infinite region. Let F be polygon that defines the finite region.

Second, let cd be a sub-segment of $w_h^*w_{h+1}^*$ that is a chord in R. Combining $R[c, d]$ with cd and $R[d, c]$ with cd, yields one finite sub-polygon that contains the segment w_{h+1}^*y (this is analogous to the above infinite case) and another finite sub-polygon that does not (this is analogous to the above finite case). Let G be the sub-polygon that does not contain w_{h+1}^*y.

Since $F, G \cap P[z, y] = \emptyset$, and since y and z are not in the interiors of any F or G, we know that $P[z, y]$ must be balanced with respect to $w_h^*w_{h+1}^*$.

(See Figure 2.21) Since $P[y, w_h^*]$ and $P[z, y]$ are both balanced with respect to $w_h^* w_{h+1}^*$, the combined chain, $P[z, w_h^*]$, is as well. Thus, $P[z, w_h^*]$ is balanced, and deos not contain more ext-left crossings than ext-rights with respect to $w_h^* w_{h+1}^*$, a contradiction. Therefore, D_h is not ext-right heavy with respect to $con(P_{h+1})$.

Thus, D_h is balanced and not ext-right heavy with respect to $con(P_{h+1})$. Since we know that D_h and D_{h+1} share the same crossings with $con(P_{h+1})$, we know that D_{h+1} is non-interfering. □

Lemma 8b. *Let D_h be non-interfering. If `ccwConnectorCut` is called on P_h, then when it terminates, D_{h+1} will be non-interfering as well.*

2.5.2 Cutting off a miss maintains non-interfering invariant

First we will show that `scanner` will correctly identify a cw-miss or a ccw-miss. Then we will show that if D_h is non-interfering, then `cwSegCut`, `ccwSegCut`, `cwRayCut`, and `ccwRayCut` terminate, and that if they cut P_h, then D_{h+1} is also non-interfering.

(See Figure 2.25) Note that if these four cutters discover a crossing with $con(P_h)$, they will not cut P_h, otherwise they will. Should such a crossing be discovered, they call `cwConnectorCut` or `ccwConnectorCut`, and then `return`. By Lemmas 7a and 7b, we know that these two cutters terminate, and by 8a and 8b, they maintain the non-interfering invariant.

Lemma 9a. *Let D_h be non-interfering. If the variable `cwExtremums` in `scanner` reaches -1 for the first time at some vertex $v_j \in P_h$, then v_j is a cw-miss in P_h.*

Proof. First we will show that v_j is a reflex minima vertex. We know that `cwExtremums` reached -1 for the first time at v_j. Thus in the step before `cwExtremums` was altered, its value was either 0 or -2. If its value was -2, then since this variable was initialized to 0, it must have already reached -1. Therefore, this variable was at 0 and was decremented when `scanner` reached the reflex minima vertex v_j.

Without loss of generality, let P_h be a candidate cw-component. This proof will be an analysis of three cases. Either v_j' is in: (1) $P[v_j, w_h]$, (2) D_h, or (3) $P[w_h^*, v_j]$. We show that the third case always happens, by showing that the first two cannot.

Case 1: $v_j' \in P[v_j, w_h]$

(See Figure 2.26) We know that $P[v_j, v_j']$ and $v_j' v_j$ forms a polygon Q . We will apply Lemma 1 to Q, but first we must ensure that Q has no horizontal

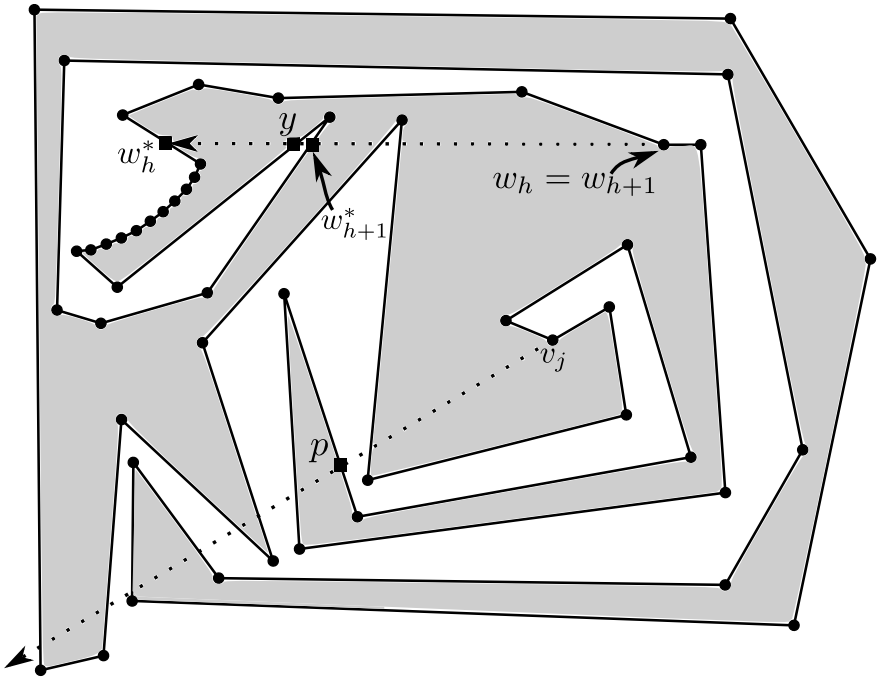

Figure 2.25: The first event to occur is for the clockwise step of `scanner` to find the cw-miss v_j. It then calls `cwMissChoose` which finds p, the closest crossing with $\overrightarrow{cw}(v_j)$ to v_j in $P[v_j, w_h]$. It then calls `cwSegCut(W,`v_j`,p)` which traverses counterclockwise from w_h^* until it crosses the segment $v_j p$. However, before it does, it crosses $con(P_h)$ at y and ignores v_j and calls `cwConnectorCut` because P_h is a candidate cw-component.

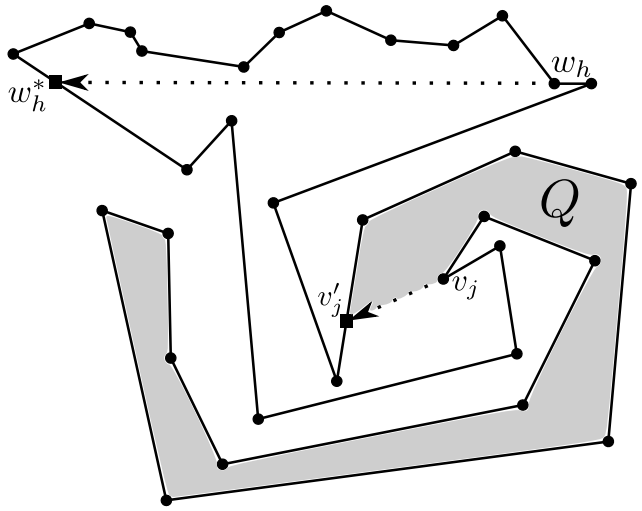

Figure 2.26: This figure is used in a contradiction proof. We assume that `scanner` found the cw-miss v_j, even though it would have chosen the counterclockwise neighbor of v_j' instead.

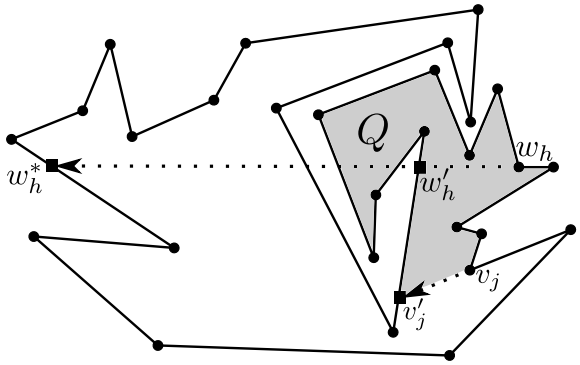

Figure 2.27: This figure is used in a contradiction proof. The function `scanner` found the cw-miss v_j. Assume that $v_j' \in D_h$.

edges. There is only one horizontal edge in P, which is the clockwise edge of w_h, but this edge is not in Q. Thus we can safely apply Lemma 1. It shows that Q has one more convex maxima than reflex minima vertex.

Note that v_j is a reflex minima in P, but not in Q. Thus, including v_j we know that $P[v_j, v_j']$ has an equal number of reflex minima and convex maxima vertices. Thus, `cwExtremums` must have first reached -1 in $P[v_j', w_h]$, a contradiction. Therefore, $v_j' \notin P[v_j, w_h]$.

Case 2: $v_j' \in D_h$

(See Figure 2.27) Again $P[v_j, v_j']$ and $v_j v_j'$ form Q. We know that $w_h \in Q$. Since `scanner` reached v_j without halting, we know that no point in $P[v_j, w_h]$ is above $con(P_h)$. Thus, we know $w_h' \notin P[v_j, w_h]$. For the same reason we know that $w_h' w_h \cap v_j' v_j = \emptyset$. Thus, $w_h' \in Q$. By process of elimination, we know that $w_h' \in Q[w_h, v_j']$. Since $Q[w_h, v_j'] \subset D_h$, we know that $w_h' \in D_h$, a contradiction by Lemma 4. Thus, $v_j' \notin D_h$.

Thus, by process of elimination we know that case (3): $v_j' \in P[w_h^*, v_j]$, must happen. Therefore, v_j is a cw-miss in P_h. □

Lemma 9b. *Let D_h be non-interfering. If the variable `ccwExtremums` in `scanner`, reaches -1 for the first time at some vertex $v_j \in P_h$, then v_j is a ccw-miss in P_h.*

In Lemma 10a, which follows, we show that `cwRayCut` either finds a satisfying crossing or

instead discovers a crossing with $con(P_h)$, calls `cwConnectorCut` or `ccwConnectorCut`, and abandons its efforts. We have already shown that `cwConnectorCut` and `ccwConnectorCut` terminate and maintain the invariant.

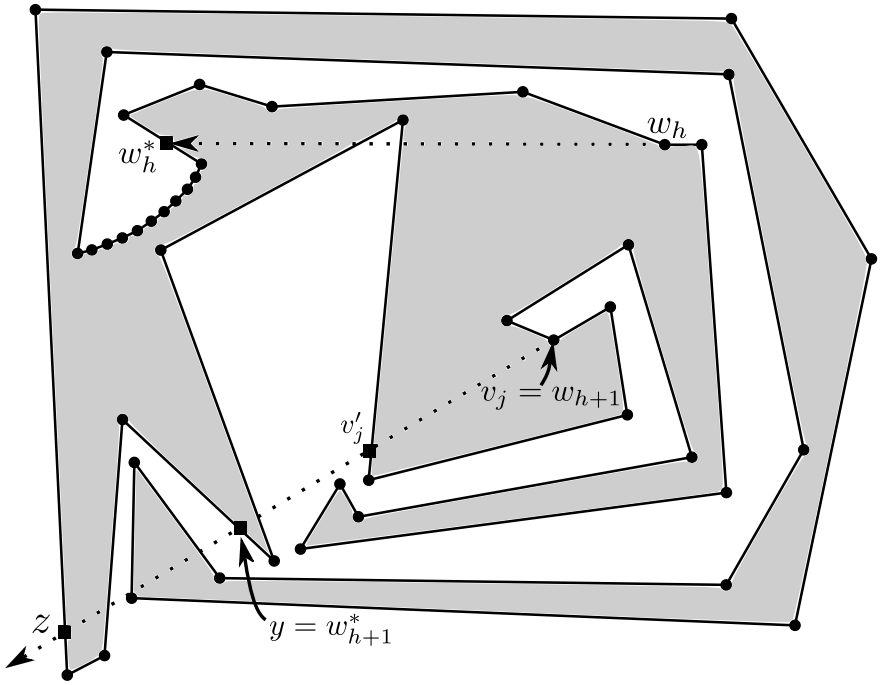

Figure 2.28: The clockwise step of `scanner` finds the cw-miss v_j. It calls `cwMissChoose` which calls `cwRayCut` which traverses counterclockwise from w_h^* until reaching y. Note that z was the first time that `crossingCounter == 1`, but at the time, it was not the closest known crossing to v_j. Thus, the walk continued until reaching y. Note that v'_j also satisfies the four conditions.

Lemma 10a. *Let D_h be non-interfering and let `scanner` find the cw-miss $v_j \in P_h$. If `cwRayCut` is called, and it does not call `cwConnectorCut` or `ccwConnectorCut`, then the counterclockwise walk of P_h from `W.cwEnd` finds a crossing $x \in P_h$ with $\overrightarrow{cw}(v_j)$ such that: (1) x is an ext-left-crossing, (2) the walk visits one more ext-left than ext-right crossing, (3) x is the closest crossing to v_j on the walk, and (4) the walk finds x before reaching v_j.*

Proof. (See Figure 2.28) Without loss of generality, let P_h be a candidate cw-component. If `cwRayCut` cuts P_h at a crossing $y \in P[w_h^*, v_j]$, we know by the conditions of the if-statement that encloses `W.delete`, that the following holds for y: A) `crossingCounter == 1`, B) `closestCrossing ==` y, and C) y is an ext-left-crossing with respect to $\overrightarrow{cw}(v_j)$. Respectively, these three conditions are equivalent to the first three conditions for x. Condition

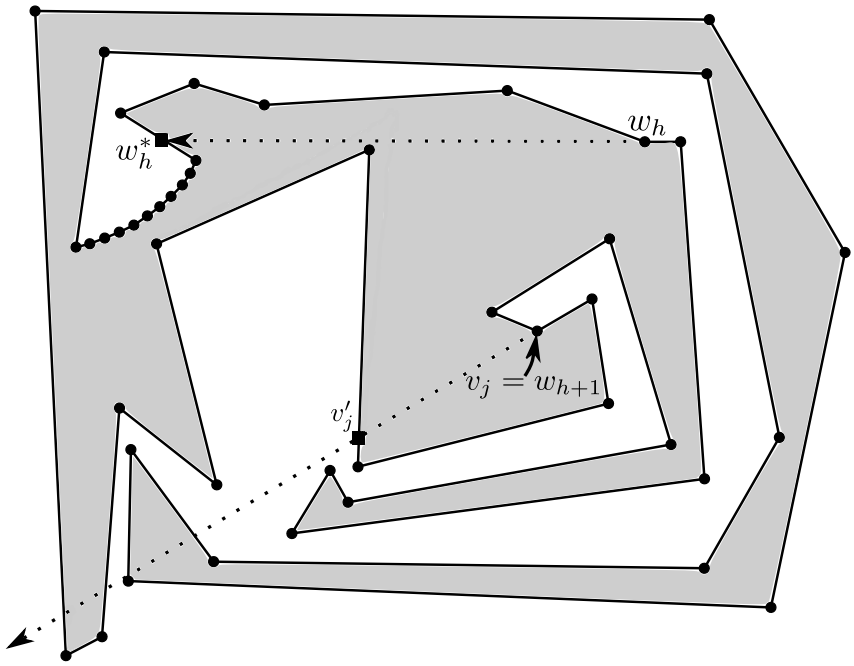

Figure 2.29: The function `cwRayCut` walks counterclockwise from w_h^* and chooses v'_j.

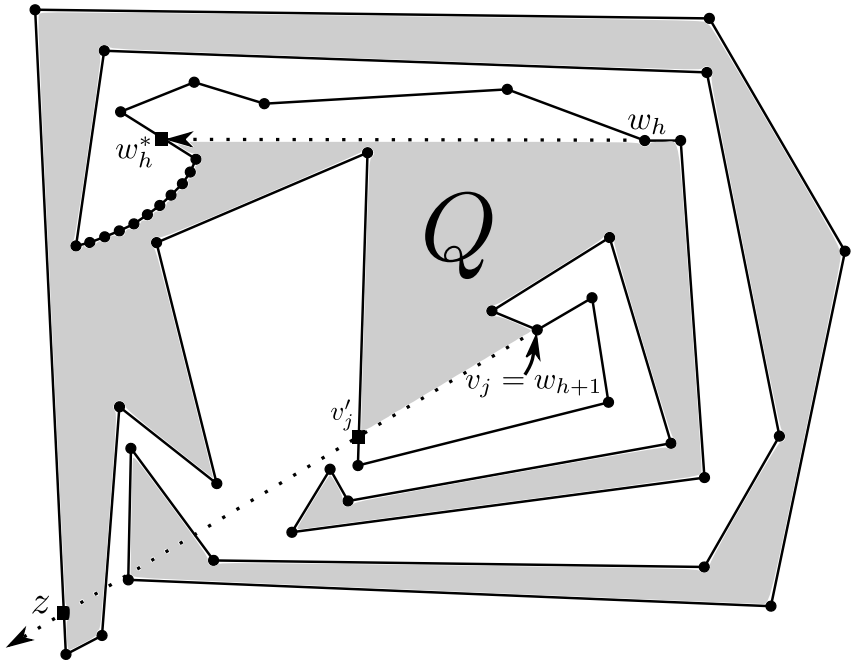

Figure 2.30: The subpolygon Q is shaded. We will consider v'_j to be an ext-left crossing in Q as it is in P. Note that the crossings with $\overrightarrow{cw}(v_j)$ that are closest (v'_j) and furthest from v_j (z) are ext-left crossings, and that there is one more ext-left crossing.

(4) is satisfied by our assumption that $y \in P[w_h^*, v_j]$.

(See Figure 2.29) Now we only need to show that there must be a crossing in $P[w_h^*, v_j]$ that satisfies these four conditions. We will show that v_j' is such a crossing. Assume that `cwRayCut` chooses v_j'. By definition, v_j' is the closest crossing to v_j. This satisfies condition (3). Since it is the closest crossing, by the Jordan Curve theorem, v_j' is an ext-left crossing. This satisfies condition (1). By Lemma 9a, $v_j' \in P[w_h^*, v_j]$. Thus, the walk reaches w_h' before reaching v_j, unless it finds another satisfying crossing first. This satisfies condition (4).

We will now show that v_j' satisfies condition (2) for x. Assume that the algorithm reaches v_j', without finding another satisfying crossing first. Thus, $P[w_h^*, v_j']$ does not cross $con(P_h)$, since if it did, `cwRayCut` would have called `cwConnectorCut`. Similarly, $P[v_j, w_h]$ cannot cross $con(P_h)$ either, since `scanner` did not call `cwConnectorCut`. Thus, the chains $P[w_h^*, v_j']$, and $P[v_j, w_h]$ do not cross $con(P_h)$.

(See Figure 2.30) Further, $P[v_j, w_h]$ cannot cross the line containing $con(P_h)$, since `cw.WalkActive` was not set to `false`. Thus v_j is below $con(P_h)$. Since v_j is a reflex minima vertex, v_j' must also be below $con(P_h)$. Thus, $v_j v_j'$ cannot cross $con(P_h)$, and since it is a chord, it cannot cross P either. Therefore, the chains $P[w_h^*, v_j']$ and $P[v_j, w_h]$, and the segments $con(P_h)$ and $v_j v_j'$ form the polygon Q.

Since `cwMissChoose` called `cwRayCut`, we know $P[v_j, w_h]$ does not cross $\overrightarrow{cw}(v_j)$. Thus, the crossings between $\overrightarrow{cw}(v_j)$ and Q are with $Q[w_h^*, v_j'] = P[w_h^*, v_j']$. Note that we will consider v_j' to be an ext-left crossing in Q as it is in P.

By the Jordan Curve theorem, if we walk along $\overrightarrow{cw}(v_j)$ from v_j, the crossings we encounter will alternate between ext-left and ext-right crossings. By the Jordan Curve theorem, the furthest crossing from v_j in Q is an ext-left-crossing. Thus, since the crossings alternate and the closest and furthest crossings to v_j are ext-left-crossings, we know that there is one more ext-left-crossing than ext-right-crossing in $Q[w_h^*, v_j'] = P[w_h^*, v_j']$. Thus, v_j' satisfies condition (2). This condition implies that when the walk reaches v_j', that `crossingCounter == 1`. Thus, v_j' satisfies the four conditions for x. □

Lemma 10b. *Let D_h be non-interfering and let `scanner` find the ccw-miss $v_j \in P_h$. If `ccwRayCut` is called, and it does not call `cwConnectorCut` or `ccwConnectorCut`, then the clockwise walk of P_h from `W.ccwEnd` finds a crossing $x \in P_h$ with $\overrightarrow{ccw}(v_j)$ such that: (1) x is an ext-right-crossing, (2) the walk visits one more ext-right than ext-left crossing, (3) x is the closest crossing to v_j on the walk, and (4) the walk finds x before reaching v_j.*

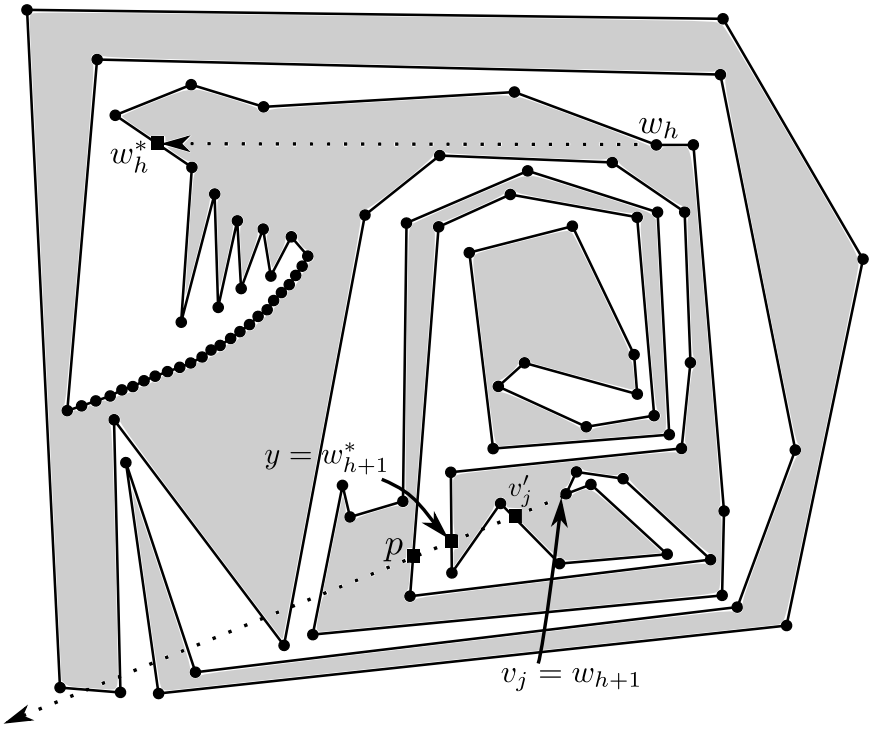

Figure 2.31: The first event to occur is for the clockwise step of `scanner` to find the cw-miss v_j. It then calls `cwMissChoose` which calls `cwSegCut(W,`v_j`,p)`, which traverses counterclockwise from w_h^* until crossing pv_j at y. Note that if the walk continued to v_j', this would also satisfy the two conditions.

Lemma 11a. *Let D_h be non-interfering, let `scanner` find the cw-miss $v_j \in P_h$, and let q be the crossing with $P[v_j, w_h]$ and $\overrightarrow{cw}(v_j)$ that is closest to v_j. If `cwSegCut` is called, and it does not call `cwConnectorCut` or `ccwConnectorCut`, then it finds a crossing $x \in P_h$ with $\overrightarrow{cw}(v_j)$ such that (1) $x \in qv_j$, (2) the walk finds x before reaching v_j, and (3) x is an ext-left-crossing with respect to $\overrightarrow{cw}(v_j)$.*

Proof. (See Figure 2.31) Without loss of generality, let P_h be a candidate cw-component. First, we will assume condition (2), that `cwSegCut` finds a crossing w_{h+1}^* and cuts P_h before reaching v_j. Later we will show that condition (2) must happen. We know by the condition of the if-statement that encloses `W.delete`, that $w_{h+1}^* \in qv_j$, so if P_h is cut, then condition (1) is satisfied.

To show condition (3), observe that $P[v_j, q]$ and v_jq forms a polygon Q, since q is the closest crossing with $\overrightarrow{cw}(v_j)$ in $P[v_j, w_h]$. Since the clockwise walk did not stop, we also

know that no point in $P[v_j, w_h]$ is above $con(P_h)$, so $w_h^* \notin Q$. Thus, by the Jordan Curve theorem, the first crossing `cwSegCut` finds in its counterclockwise walk of P_h from w_h^* with qv_j must be an ext-left-crossing with respect to $\overrightarrow{cw}(v_j)$, which satisfies condition (3).

Now we only need to show that `cwSegCut` always finds a crossing with qv_j in $P[w_h^*, v_j]$ at which to cut. We will show that v_j' is such a crossing. By Lemma 9a, we know that v_j is a cw-miss in P_h and that $v_j' \in P[w_h^*, v_j]$. Thus, we know that the algorithm which walks counterclockwise from w_h^* reaches v_j' before reaching v_j. This satisfies condition (2).

Because $q \in P[v_j, w_h]$ and $v_j' \in P[w_h^*, v_j]$, we know that $q \neq v_j'$. Since v_j' is the closest crossing in P to v_j, we know that $v_j' \in qv_j$. This satisfies condition (1). Thus we know that the counterclockwise walk from w_h^* crosses qv_j before reaching v_j. □

Lemma 11b. *Let D_h be non-interfering, let `scanner` find the ccw-miss $v_j \in P_h$, and let q be the crossing with $P[w_h, v_j]$ and $\overrightarrow{ccw}(v_j)$ that is closest to v_j. If `ccwSegCut` is called, and it does not call `cwConnectorCut` or `ccwConnectorCut`, then it finds a crossing $x \in P_h$ with $\overrightarrow{ccw}(v_j)$ such that (1) $x \in qv_j$, (2) the walk finds x before reaching v_j, and (3) x is an ext-right-crossing with respect to $\overrightarrow{cw}(v_j)$*

Lemma 12a. *Let D_h be non-interfering, and let `scanner` find the cw-miss $v_j \in P_h$. When `cwRayCut` or `cwSegCut` terminate, D_{h+1} will be non-interfering as well.*

Proof. Without loss of generality, let P_h be a candidate cw-component. By Lemma 9a, we know that v_j is a cw-miss in P_h, and that $v_j' \in P[w_h^*, v_j]$. The function `scanner` called `cwMissChoose` which called either cutter `cwRayCut` or `cwSegCut`. We know that `cwRayCut` and `cwSegCut` terminate by Lemmas 10a and 11a, respectively. The cutters may terminate either upon finding the crossing w_{h+1}^* or upon crossing $con(P_h)$. Note that we will handle both cutters in this proof.

If either cutter, `cwRayCut` or `cwSegCut`, cross $con(P_h)$ in their walk, then it calls the cutter `cwConnectorCut`. By Lemma 7a, we know that `cwConnectorCut` halts and by Lemma 8a that the resulting D_{h+1} is non-interfering. Thus, we need to show that if `cwRayCut` or `cwSegCut` do not cross $con(P_h)$, then the resulting D_{h+1} is non-interfering.

$P[w_h^*, w_{h+1}^*]$ does not cross $con(P_h)$:

> Since `cwConnectorCut` maintains the non-interfering invariant, for the duration of this proof we will assume that $P[w_h^*, w_{h+1}^*]$ does not cross $con(P_h)$.

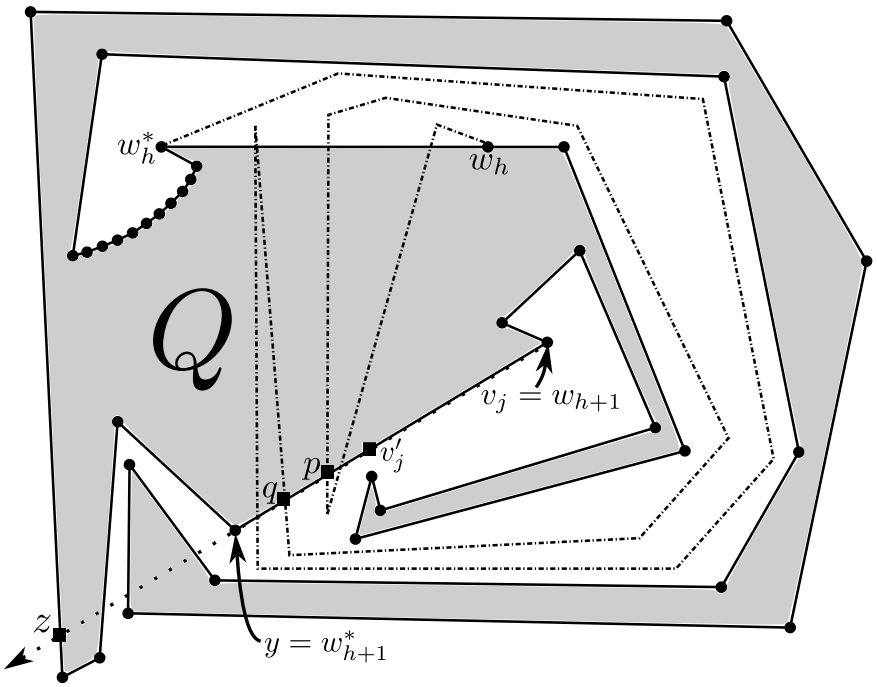

Figure 2.32: The polygon Q is shaded, and D_h is dashed. The point $p \in D_h$ is the first ext-right heavy crossing with respect to $con(P_{h+1})$ that we find in a clockwise walk from w_h^*, and q is the previous ext-right-crossing.

$P[v_j, w_h]$ **does not cross** $con(P_h)$:

We know this because `scanner` did not call `cwConnectorCut`.

$P[v_j, w_h]$ **does not cross** $con(P_{h+1})$:

(See Figure 2.28) If `cwRayCut` was called, then `cwMissChoose` determined that $P[v_j, w_h]$ does not cross $con(P_{h+1})$. (See Figure 2.31) Otherwise, if `cwSegCut` was called, then by Lemma 11a, we know that the crossing it finds, w_{h+1}^*, with $\overrightarrow{cw}(v_j)$ is strictly closer to v_j than any crossing `scanner` and `cwMissChoose` visited in $P[v_j, w_h]$.

$P[w_h^*, w_{h+1}^*]$ **does not cross** $con(P_{h+1})$:

By Lemma 10a, `cwRayCut`, w_{h+1}^* is the crossing between $P[w_h^*, w_{h+1}^*]$ and $\overrightarrow{cw}(v_j)$ that is closest to v_j. The same holds for `cwSegCut`.

(See Figure 2.30) Since $P[w_h^*, w_{h+1}^*]$ and $P[v_j, w_h]$ do not cross $con(P_h)$ and $con(P_{h+1})$, these two chains and two segments form the polygon Q.

We will walk D_h clockwise from w_h^* and as we walk, we may cross in and out of Q by crossing $con(P_h)$ or $con(P_{h+1})$. Note that this walk is not done by an algorithm, it is simply part of the proof. If we cross into Q by crossing $con(P_h)$, we will consider this crossing to be a in_h crossing. In a similar way, we also define out_h, in_{h+1} and out_{h+1} crossings. Also, at any given point, we will consider the number of in_h crossings to be $|in_h|$. Let $|out_h|$, $|in_{h+1}|$ and $|out_{h+1}|$ be similarly defined.

We know that D_h is non-interfering with respect to $con(P_h)$, but assume that D_{h+1} is interfering with respect to $con(P_{h+1})$. Thus, either D_{h+1} is ext-right heavy or not balanced with respect to $con(P_{h+1})$. Note that by the construction of Q, the chains D_{h+1} and D_h both have the exact same crossings with $con(P_h)$ and $con(P_{h+1})$. Therefore, we will prove this property for D_h instead.

D_h is not ext-right heavy with respect to $con(P_{h+1})$

(See Figure 2.32) Assume for the purposes of contradiction that D_h is ext-right heavy with respect to $con(P_{h+1})$. We will show that then it is ext-right heavy with respect to $con(P_h)$ as well, a contradiction.

Let $p \in D_h$ be an out_{h+1} crossing that is the first ext-right heavy crossing with respect to $con(P_{h+1})$ found in a clockwise walk from w_h^*. Let $q \in D_h$ be the previous out_{h+1} crossing. Since p is the first ext-right heavy crossing, q must also be ext-right, because if it were ext-left, then p would not be the first that is ext-right heavy.

Thus, $P[q, w_h^*]$ is balanced with respect to $con(P_{h+1})$ and thus at this time, $|out_{h+1}| = |in_{h+1}|$. By the Jordan Curve theorem, we know that each in_h crossing is followed by either an out_h or out_{h+1} crossing. Thus, immediately after reaching q, we know that $|in_h| \leq |out_h| + |out_{h+1}|$. Similarly, $|out_h| \leq |in_h| + |in_{h+1}|$. By this equality and these two inequalities, we know that at this time, $|in_h| = |out_h|$. Also, at this time we are outside of Q.

Since p is the first ext-right heavy crossing, and since we are currently outside of Q, if we continue our clockwise walk of D_h from q, there can be no other out_{h+1} or in_{h+1} crossings until we reach p. Since p is an out_{h+1} crossing, we must have crossed into Q with an in_h crossing first. Thus when we reach p, we know that $|in_h| > |out_h|$, which implies that D_h is ext-right heavy with respect to $con(P_h)$. Therefore, D_h is interfering, a contradiction. Thus D_h is not ext-right heavy

with respect to $con(P_{h+1})$.

D_h is balanced with respect to $con(P_{h+1})$

Assume for the purposes of contradiction that D_h is not balanced with respect to $con(P_{h+1})$. We know that $|in_{h+1}| \neq |out_{h+1}|$ for D_h. Also, since D_h is non-interfering, we know that $|in_h| = |out_h|$. Since both endpoints of D_h are not in the interior of Q, the number of times the clockwise walk of D_h from w_h^* to w_h enters Q must be the same as the number of times it leaves Q. Thus, $|in_{h+1}| + |in_h| = |out_{h+1}| + |out_h|$, a contradiction. Thus, D_h is balanced with respect to $con(P_{h+1})$.

$con(P_{h+1})$ is a pseudochord

Since v_j is a cw-miss, we know that in the neighborhood of v_j, the segment $con(P_{h+1})$ is in the interior of P. If `cwRayCut` found w_{h+1}^*, then by Lemma 10a, w_{h+1}^* is an ext-left-crossing with respect to $\overrightarrow{cw}(v_j)$. Otherwise if `cwSegCut` found w_{h+1}^*, then by Lemma 11a, we know that w_{h+1}^* is an ext-left-crossing with respect to $\overrightarrow{cw}(v_j)$. Thus, either way, w_{h+1}^* is an ext-left-crossing with respect to $\overrightarrow{cw}(v_j)$. Therefore, in the neighborhood of w_{h+1}^*, the segment $con(P_{h+1})$ is in the interior of P. Thus, $con(P_{h+1})$ is a pseudochord.

Thus $con(P_{h+1})$ is a pseudochord, and D_h is balanced and not ext-right heavy with respect to $con(P_{h+1})$. Since D_h and D_{h+1} have the exact same crossings with $con(P_{h+1})$, we know that D_{h+1} is non-interfering. □

Lemma 12b. *Let D_h be non-interfering, and let `scanner` find the ccw-miss $v_j \in P_h$. When `ccwRayCut` or `ccwSegCut` terminate, D_{h+1} will be non-interfering as well.*

2.5.3 The algorithm maintains our non-interfering invariant

We have shown that each time `scanner` finds a cw-miss, ccw-miss, or a crossing with $con(P_h)$, we will eventually cut P_h to yield P_{h+1}. The following lemma finally shows that each P_h the algorithm generates will maintain the non-interfering invariant.

Lemma 13. *The algorithm maintains the non-interfering invariant.*

Proof. We will use weak induction on the number of times the algorithm cuts some P_i to P_{i+1}. Basis: Since P_0 is a component, by Lemma 2, D_0 is non-interfering.

Inductive step: Assume that D_h is non-interfering. We will show that D_{h+1} is as well. There are six function calls that may cut P_h into P_{h+1}, namely `cwConnectorCut`, `ccwConnectorCut`, `cwRayCut`, `ccwRayCut`, `cwSegCut` and `ccwSegCut`. Respectively, Lemmas 7a, 7b, 10a, 10b, 11a and 11b show that these functions halt.

Lemmas 8a and 8b show that if `cwConnectorCut` or `ccwConnectorCut` cut P_h into P_{h+1}, that D_{h+1} is non-interfering. Lemmas 12a and 12b show the same for the cutters: `cwRayCut`, `ccwRayCut`, `cwSegCut` and `ccwSegCut`. Thus, we have shown that all six cutters terminate and maintain the invariant. Therefore, D_{h+1} will be non-interfering and the invariant will be satisfied. □

2.6 Correctness and running time analysis

In this section we will show that `getReflexMinFreeComp` finds a reflex-minima-free component in $O(n)$ time. First we will show that it terminates in $O(n)$ time. Then Lemma 15 shows that it returns a reflex-minima-free component.

Lemma 14. *Given P as input, where P has n vertices, `getReflexMinFreeComp` runs in $O(n)$ time.*

Proof. First, `getReflexMinFreeComp` finds P_0 in P in $O(n)$ time. Then it passes P_0 to `scanner`. We use an amortized analysis to show that each time the algorithm removes z vertices from a candidate component P_h, to obtain P_{h+1}, that it visited $O(z)$ vertices. Without loss of generality, let P_h be a candidate cw-component.

Only `cwConnectorCut`, `cwRayCut`, `ccwRayCut`, `cwSegCut` and `ccwSegCut` may cut P_h into P_{h+1}. Respectively Lemmas 7a, 10a, 10b, 11a, and 11b show that these functions halt.

Let the clockwise walk of `scanner` visit j vertices before finding a crossing $p \in con(P_h)$. Because `scanner` alternates between the clockwise and counterclockwise walks, we know that the counterclockwise walk of `scanner` also visited $j \pm 1$ (we will ignore the ± 1) vertices, the last one being v_g. Thus, `scanner` visit $2j$ vertices.

Next, `scanner` calls `cwConnectorCut` which walks counterclockwise from w_h^* past v_g until it finds w_{h+1}^*. It visits k vertices. We know that $k > j$ because otherwise the counterclockwise walk of `scanner` would have found a crossing before the clockwise walk found p. Altogether,

`scanner` and `cwConnectorCut` visits $2j + k$ vertices and removes k vertices. Note that $2j + k < 3k \in O(k)$.

Let the counterclockwise walk of `scanner` visit f vertices before finding a crossing $z \in con(P_h)$. This case is nearly identical to the above case. The clockwise walk of `scanner` also visited f vertices and then `cwConnectorCut` walks counterclockwise from w_h^* past z, visiting e vertices, where $e > f$. We remove e vertices and $2f + e < 3e \in O(e)$.

Let the clockwise walk of `scanner` find the cw-miss v_j and let $P[v_j, w_h]$ have a vertices. The counterclockwise walk of `scanner` halted after also visiting a vertices. The clockwise walk of `scanner` calls `cwMissChoose` which walks $P[v_j, w_h]$ again, visiting another a vertices. Thus far `scanner` and `cwMissChoose` have visited $3a$ vertices. Then this function calls either `cwRayCut` or `cwSegCut`. We will handle these two cutters together.

First assume that whichever cutter is chosen finds w_{h+1}^* without calling `cwConnectorCut`, after visiting b vertices. Note that we do not know if $a > b$ or $b > a$. We will remove the chains $P[v_j, w_h]$ and $P[w_h^*, w_{h+1}^*]$. Thus, we remove $a+b$ vertices and visit $3a+b \in O(a+b)$.

Now assume that `cwMissChoose` called a cutter which crossed $con(P_h)$ after visiting c vertices. So far, `scanner` visited $2a$ vertices, `cwMissChoose` visited another a vertices, then which ever cutter was called visited c vertices. This is $3a+c$ vertices in total. We know that $c > a$ because otherwise the counterclockwise walk of `scanner` would have crossed $con(P_h)$ before the clockwise walk found v_j.

Upon finding crossing $con(P_h)$, the cutter calls `cwConnectorCut` which again walks counterclockwise from w_h^* past this crossing to find $w_{h+1}^* \in con(P_h)$. It will visit d vertices and since the walk passes this crossing, we know that $d > c$. We remove d vertices and visit $3a + c + d < 5d \in O(d)$ vertices.

Note that if the counterclockwise walk found a ccw-miss and then the clockwise search for w_{h+1}^{**} found a crossing with $con(P_h)$ instead, the argument would be symmetric.

We have shown that if the algorithm cuts P_h to get P_{h+1}, it visits a number of vertices that is on the order of the number of vertices it removes. Thus, `getReflexMinFreeComp` takes $O(n)$ time, and if P_0 has m vertices, then the algorithm visits $O(m)$ vertices before terminating. Thus, altogether the algorithm takes $O(n)$ time. □

In this final lemma, we will show that eventually, for some candidate component P_h, `scanner` will not find a crossing with $con(P_h)$, a cw-miss, or ccw-miss. Thus, no cutter will cut P_h. We will show that P_h is a reflex-minima-free component.

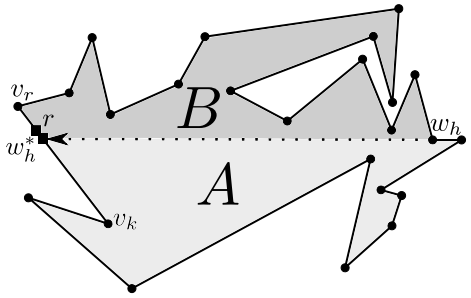

Figure 2.33: The chains P_h and D_h do not cross $con(P_h)$. The polygon A is shaded lighter than B. The chain D_h contains one more convex maxima than reflex minima, and P_h contains exactly as many reflex minimas as convex maximas. In this figure, P_h is not reflex-minima-free. The vertex v_k is a reflex minima and it's counterclockwise neighbor is a convex maxima. Note that the clockwise walk of `scanner` will not call a cutter, but the counterclockwise walk will upon reaching the ccw-miss v_k.

Lemma 15. *Given P, the function `getReflexMinFreeComp` outputs a reflex-minima-free component.*

Proof. Lemma 14 shows that `getReflexMinFreeComp` halts. Let P_h be the final candidate component that `scanner` processed. Without loss of generality, let P_h be a candidate cw-component. By Lemma 13, we know that D_h is non-interfering. We also know that P_h does not cross $con(P_h)$, since if it did, `scanner` would have called `cwConnectorCut` instead and produced P_{h+1}.

For the purposes of contradiction, assume that D_h crosses $con(P_h)$. Since P_h does not cross $con(P_h)$, we know that P_h and $con(P_h)$ forms a polygon A. Thus, if we walk clockwise from w_h^*, the first crossing with $con(P_h)$ must be an ext-right-crossing. Thus, D_h is ext-right heavy, a contradiction. Thus the pseudochord $con(P_h)$ is a chord, and P_h is a component. Now, we will show that P_h is reflex-minima-free.

(See Figure 2.33) Let the polygon bound by D_h and $con(P_h)$ be B. Let v_r be the clockwise neighbor of w_h^*, and let $r \in w_h^* v_r$ be ϵ distance from w_h^*. If ϵ is small enough, $P[w_h, r]$ will not cross rw_h since D_h did not cross $con(P_h)$. To enable us to apply Lemma 1 to B, we replace $con(P_h)$ with $w_h r$. By the lemma, we know that D_h has one more convex maxima than reflex minima.

Applying Lemma 1 to P also shows that P has one more convex maxima than reflex minima. Thus by the Pigeon Hole principle, A, and thereby P_h has exactly as many reflex minimas as convex maximas.

For the purposes of contradiction, assume that P_h is not reflex-minima-free. Thus it contains at least one reflex minima vertex. Then, since P_h contains as many reflex minimas as convex maximas, we know P_h also contains exactly as many convex maximas.

Without loss of generality, assume that in the clockwise walk of `scanner`, the variable `cwExtremums` never goes below 0. This variable is incremented each time the clockwise walk of `scanner` encounteres a convex maxima, and decremented each time it reaches a reflex minima. Since P_h contains as many convex maximas as reflex minimas, we know that at the end of this walk, `cwExtremums == 0`.

Therefore, we know that the last time `cwExtremums` was changed in this walk, it must have been must decremented upon reaching some reflex minima vertex v_k. If it was incremented instead, then it would have originally been below 0. Thus, v_k is a reflex minima vertex, and there are no reflex minima or convex maxima vertices in $P[w_h^*, v_k] \setminus v_k$.

Therefore, in the counterclockwise walk of `scanner`, v_k is the first time `ccwExtremums` is changed. Since v_k is a reflex minima, `ccwExtremums` becomes -1 and `scanner` calls `ccwMissChoose`, which will in turn call a cutter that cuts P_h to P_{h+1}, a contradiction. Thus, since P_h is the final candidate component processed by `scanner`, we know that it must be a reflex-minima-free component. □

We have already proven the following theorem. Note that we will spend $O(n)$ time to find P_0.

Theorem 1. *Given P, which contains n vertices, the algorithm returns a reflex-minima-free component in $O(n)$ time.*

Chapter 3

Algorithm to find nonredundant component

In this chapter, we present Algorithm 9, which refers to the function `getNonRedundantComp`. This function takes as input a general polygon P and returns a nonredundant component.

This function first calls `getReflexMinFreeComp` which takes a general polygon P as input and returns a reflex-minima-free component with the help of `scanner`. Then if this component is clockwise, it triangulates it with Algorithm 7, which is the function `cwTrapezoidation`. Otherwise, it is counterclockwise and it uses `ccwTrapezoidation` (which is symmetric to the clockwise case and thus omitted).

Note that `cwTrapezoidation` and `ccwTrapezoidation` do not actually compute the triangulation of the component, but instead the trapezoidation, also known as horizontal visibility map. However, Fournier and Montuno in [15] as well as Chazelle and Incerpi in [8], show that you can convert a trapezoidation to a triangulation in linear-time. Furthermore, in [1], the authors state that "every published triangulation algorithm has concentrated on improving the running time of producing a trapezoidation of a simple polygon."

This algorithm obtains the trapezoidation of a reflex-minima-free component in linear-time. It is a very simple algorithm and uses only a single clockwise walk and a single stack.

A component is said to be *cw-nonredundant* if it is a cw-component and does not entirely contain another cw-component. A *ccw-nonredundant* is similarly defined.

The reflex-minima-free component and it's triangulation then becomes input for Algorithm 8, which is the function `getCwNonRedundantComp`. This algorithm is the central algorithm of this chapter. The function `getCwNonRedundantComp` returns a cw-nonredundant component.

This cw-nonredundant component may not be reflex-minima-free. Thus, we call `scanner` on it which will return a reflex-minima-free subcomponent if it exists. The resulting component, if it exists, must be counterclockwise, since a cw-nonredundant component cannot contain another clockwise component. However, if the original polygon was already reflex-minima-free, then `w` will still store a clockwise component.

We again obtain the trapezoidation of the component stored in `w` with either the functions `cwTrapezoidation` or `ccwTrapezoidation`. The triangulation based on this trapezoidation is used as input to `getCcwNonRedundantComp` which returns a ccw-nonredundant subcomponent.

We will show that each step of `getNonRedundantComp` takes an amount of time proportional to the given chain. Thus, it takes $O(n)$ time to complete.

The function `getCwNonRedundantComp` is based on the main algorithm from the paper *LR-Visibility in Polygons* by Das, Heffernan and Narasimhan [11]. This paper uses some very heavy machinery including Chazelle's very complex linear-time triangulation algorithm [7], and Guibas, et al. [17] which computes a shortest path tree.

In Das, et al. [11], Chazelle's algorithm is used only because Guibas, et al. [17] requires a triangulation in order to obtain a shortest path tree in linear-time. However, since we can reduce a polygon, or a component to a reflex-minima-free component, we are able to triangulate it easily with `cwTrapezoidation` or `ccwTrapezoidation` and avoid triangulating a general polygon.

3.1 Obtain trapezoidation of reflex-minima-free polygon

The triangulations of many classes of polygons can be computed by simple algorithms in linear-time. For instance monotonic polygons, palm polygons [13], and star-shaped polygons [15]. Below is another very simple algorithm that triangulates the class of reflex-minima-free components, where the component has m vertices, in $O(m)$ time.

In this chapter, without loss of generality, we will assume that we are given the reflex-minima-free clockwise component P_h. We will draw trapezoidation edges that will be parallel to $con(P_h)$, that is, they will be horizontal.

First, consider the vertex $v_k \in P_h$. Either both v_{k-1} and v_{k+1} are above it, both are below

it, v_{k-1} is below and v_{k+1} is above, or v_{k+1} is below and v_{k-1} is above. Respectively, these four classes partition the set of vertices into: *minima vertices*, *maxima vertices*, *ext-right vertices*, and *ext-left vertices*. Each of these four classes can be broken into two sub-classes depending on if it is convex or reflex. For instance, if a minima vertex is convex, then it is a *convex minima vertex*.

This algorithm does a single clockwise walk and uses a single stack. As the walk goes downwards, it pushes vertices onto the stack. Eventually, since P_h is reflex-minima-free, it will make a right turn and walk upwards. If it made a left turn, P_h would contain a reflex minima vertex. As it walks upwards it shoots rays to the left from the vertices on the stack that are below the current vertex, if any, and then shoots one to the right from the current vertex.

Note that we obtain these trapezoid edges in traversal order, so there is no need to sort them at completion. Also note that because of our general position assumption, a trapezoidation ray shot horizontally from one vertex cannot hit another vertex.

Algorithm 7 cwTrapezoidation (Chain W)

```
Stack s = ∅
for each vertex x ∈ W in cw order from W.cwNeigh(W.ccwEnd) to W.cwEnd do
    if x is an ext-left or reflex maxima vertex then
        while x is above s.peek() do
            shoot trap. edge to left from s.peek() onto W.ccwEdge(x)
            s.pop()
        end while
        shoot trap. edge to right from x onto the edge composed of s.peek() and the
        vertex below it on the stack s
    end if
    if x is an ext-right or reflex maxima vertex then
        s.push(x)
    end if
end for
```

Lemma 16. *The trapezoidation edges drawn by* `cwTrapezoidation` *form a proper trapezoidation.*

Proof. First, we know that only ext-right vertices and reflex maxima vertices are pushed onto the stack. When these vertices are popped, the algorithm shoots a trapezoidation ray from each vertex to the left. Also, all ext-left and reflex maxima vertices shoot trapezoidation

rays to the right.

As the clockwise traversal walks downwards, we add push the ext-right vertices we encounter to the stack. Note that the vertices on the stack are ordered by their height. Eventually we will walk upwards. Since we are walking upwards, and the stack is also in height order, we can accurately connect the vertices visited in this upwards walk with the corresponding vertices stored on the stack. Consider a trapezoidation edge that the algorithm generates, yv_y, where without loss of generality, v_y is an ext-left vertex. We know that $P[v_y, y]$ and the segment yv_y form a polygon Q. Therefore, if P_h crossed yv_y, then the lowest vertex $v_l \in P_h$ that is in the interior of Q would be a reflex minima vertex, a contradiction.

Note that every vertex of P_h must be below $con(P_h)$, otherwise the component would contain a reflex minima vertex. Since our walk starts and ends at the same height as $con(P_h)$ and never goes above it, we know that we must have encountered an edge containing the hit-point of each vertex on the stack. Thus at completion, the stack must be empty. □

Lemma 17. *Given a component P_h with m vertices, the function* `cwTrapezoidation` *runs in $O(m)$ time.*

Proof. We do one single walk, which takes linear-time. During this walk, we push some vertices onto the stack and then pop them off, and we never look beneath the top two elements of the stack. Therefore, maintaining the stack takes amortized linear-time. □

We have proved the following:

Theorem 2. *The function* `cwTrapezoidation` *gives a proper trapezoidation of a reflex-minima-free component in linear-time.*

3.2 Shortest paths, shortest path trees, and order queries

In this section we define shortest paths and then shortest path trees. Shortest path trees are the basis for order queries, the powerful tool that enable the functions `getCwNonRedundantComp` and `getCcwNonRedundantComp` as well as the main algortihm of [11].

(See Figure 3.1) Let the point $p \in P$ be on the edge $v_iv_{i+1} \subset P$, let the ray $\overrightarrow{r}$ be shot from p, and let the ray contain a point a where $a \neq p$. The ray $\overrightarrow{r}$ is *shot into the interior* of P if the sequence v_i, p, a is a left turn. Similarly, let the ray $\overrightarrow{s}$ be shot from $v_s \in P$ and

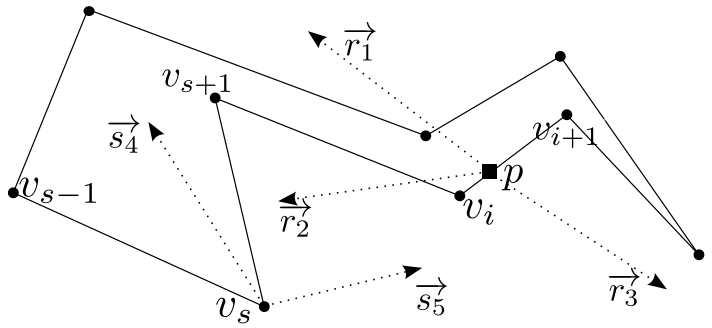

Figure 3.1: The rays $\overrightarrow{r_1}$, $\overrightarrow{r_2}$, and $\overrightarrow{s_4}$ are shot into the interior and $\overrightarrow{r_3}$ and $\overrightarrow{s_5}$ are not.

let $a \in \overrightarrow{s}$ and $a \neq v_s$. The ray $\overrightarrow{s}$ is shot into the interior of P if the sequence v_{s-1}, v_s, a is a left turn and if v_{s+1}, v_s, a is a right turn.

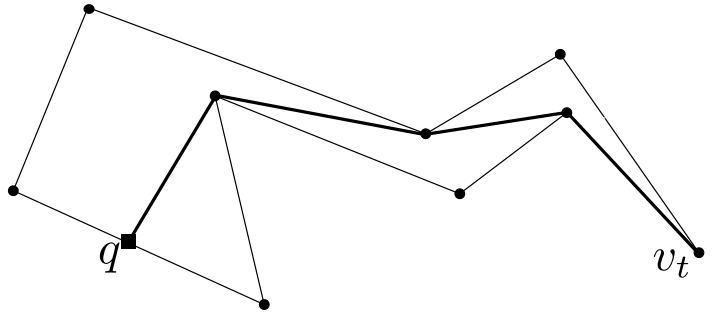

Figure 3.2: The thick black path is $SP(q, v_t)$

(See Figure 3.2) For the points $p, q \in P$, the *shortest path* from p to q, denoted $SP(p, q)$, is an open chain with endpoints p and q with the following properties: (1) each of the segments of this chain is a chord in P, (2) each segment in this chain, except for the first and last, connects two vertices of P, and (3) the length of the path is less than any other such path. Note that shortest paths are unique. Since the endpoints of a shortest path may be vertices or points, we will refer to the points and/or vertices that define the points in a shortest path as *nodes*.

(See Figure 3.3) Let the *shortest path tree* of P rooted at the point q, denoted $SPT_P(q)$, contain the shortest path from q to every vertex of P. Note that each $p \in P$ has a parent in $SPT_P(q)$. The *parent* of p is the node of the tree $SPT_P(q)$ that can see p and that the shortest path from this node to q is shorter than any other such node. Recall that Guibas, et al. [17] computes a shortest path tree in $O(n)$ time.

Heffernan in [19] and Heffernan et al. in Lemma 2 of [11], shows that we can use $SPT_P(q)$

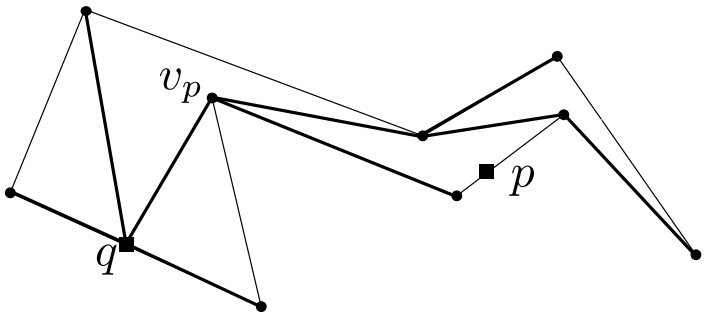

Figure 3.3: The thick black tree shows $SPT_P(q)$. That is, the shortest path tree of P rooted at q. Note that v_p is the parent of p and q is the parent of v_p in $SPT_P(q)$.

to run an *order query* in $O(1)$ time that tells us whether a bullet shot into the interior of P from p, lies in $P[q, p]$ or $P[p, q]$. We will refer to this as an *order query*.

Let $\overrightarrow{r}$ be a ray shot from p into the interior of P and let a be a point on this ray where $a \neq p$. Lemma 2 of [11] shows that if v_p, p, a is a left turn, then the hitpoint of this ray is in $P[p, q]$ and if it is a right turn then the hit point is in $P[q, p]$. We will never run an order query in which a ray is shot at a vertex.

If p is a vertex, then this is based on the parent of v in $SPT_P(q)$, which can be obtained in $O(1)$ time upon accessing a vertex of $SPT_P(q)$. Otherwise, this is based on the parent of the segment that p lies in. Note that there are $O(n)$ of these segments. In this chapter we will only run order queries from vertices.

3.3 Find cw-nonredundant subcomponent in a reflex-minima-free component

The function `getCwNonRedundantComp` detailed below in Algorithm 8 will take a component P_h with m vertices as input and output a cw-nonredundant subcomponent in $O(m)$ time. The counterclockwise case is symmetric and is ommitted.

Now we will prove the correctness of the algorithm in two cases, one if P_h is clockwise and the other if it is counterclockwise. Note that a counterclockwise component can be cw-nonredundant. Then we will show the $O(m)$ running time of this algorithm.

Lemma 18a. *Let P_h be a clockwise component, and let v_d be first reflex vertex found in a counterclockwise walk of P_h from w'_h where $v'_d \in P[w'_h, v_d]$, if such a vertex exists. If there is such a vertex, v_d, then $cw(v_d)$ is cw-nonredundant, otherwise P_h is cw-nonredundant.*

Algorithm 8 getCwNonRedundantComp (Chain W, Triangulation of W)

compute $SPT_{P_h}(W.cwEnd)$ and store parent of each vertex in W

for each reflex vertex $x \in W$ in ccw order starting at W.cwEnd **do**
  if order query shows that $x' \in W[W.cwEnd, x]$ **then**
    search all of $W[W.cwEnd, x]$ to find x'
    W.delete(W.cwEnd, x')
    W.delete(x, W.ccwEnd)
    return W
  end if
end for

return W {W already stored a cw-nonredundant component}

Proof. Assume that there exists a component $cw(v_y)$, such that $cw(v_y) \subset cw(v_d)$. We know that $v_y \in cw(v_d)$ and that $v'_y \in P[v'_d, v_y]$. Therefore, $v'_y \in P[w'_h, v_y]$, so v_d was not the first vertex found with this property, a contradiction. Now assume that the walk found no such vertex. Then there is no clockwise component inside of P_h, so it is already cw-nonredundant. □

Lemma 18b. *Let P_h be a counterclockwise component, and let v_d be first reflex vertex found in a counterclockwise walk of P_h from w_h where $v'_d \in P[w_h, v_d]$, if such a vertex exists. If there is such a vertex, v_d, then $cw(v_d)$ is cw-nonredundant, otherwise P_h is cw-nonredundant.*

Lemma 19. *Given component P_h with m vertices, and a triangulation of P_h, the functions* `getCwNonRedundantComp` *and* `getCwNonRedundantComp` *run in $O(m)$ time.*

Proof. Computing $SPT_{P_h}(W.cwEnd)$ by [17] takes $O(m)$ time. Assume that the function finds v_x and returns $cw(v_x)$. To find v_x, the function did no more than walk each vertex of P_h, which is $O(m)$ vertices. Thus it performed no more than $O(m)$ order queries, which each take $O(1)$ time. Finally upon finding v_x, if v_x, it traverses the polygon at most one more time to find v'_x, which again takes $O(m)$ time.

If the function did not find such a vertex, and simply returned P_h, then it visited each of the $O(m)$ vertices in P_h and ran at most $O(m)$ order queries. Also for a total of $O(m)$ time.

Note that traversing $SPT_{P_h}(W.cwEnd)$ in polygon order might take more than $O(1)$ time in between two adjacent vertices of P_h. Thus, we will assume that the data structure Chain also stores a pointer for each vertex to it's parent in $SPT_{P_h}(W.cwEnd)$. □

3.4 Algorithm to obtain nonredundant component from a polygon in $O(n)$ time

Finally, we show the algorithm that uses all the prior algorithms to find a nonredundant component of P.

Algorithm 9 getNonRedundantComp (Chain V)

```
Chain W = getReflexMinFreeComp(V)
{W stores a reflex-minima-free component}

if W stores a cw-component then
    Triangulation of W = cwTrapezoidation(W)
else
    Triangulation of W = ccwTrapezoidation(W)
end if
W = getCwNonRedundantComp(W, Triangulation of W)
{now W stores a cw-nonredundant component}

W = scanner(W)
{now W stores a reflex-minima-free cw-nonredundant component, which may be clockwise
or counterclockwise}

if W stores a cw-component then
    Triangulation of W = cwTrapezoidation(W)
else
    Triangulation of W = ccwTrapezoidation(W)
end if
W = getCcwNonRedundantComp(W, Triangulation of W)
{now W stores a nonredundant component}

return W
```

We have already proven the following theorem:

Theorem 3. *Given a polygon P with n vertices,* `getNonRedundantComp` *returns a nonredundant component in $O(n)$ time.*

Chapter 4

Future work

Our algorithm finds any single nonredundant component in linear-time without a general linear-time triangulation algorithm. The algorithm of [11] finds all of the components in $O(kn)$ time, where k is the number of disjoint nonredundant components, but with the use of a general linear-time triangulation algorithm.

The k factor in their algorithm exists because each time they find a disjoint nonredundant component, they compute a new shortest path tree. Perhaps it is possible to drop or lower the k factor by recycling the original shortest path tree, and/or building many small ones.

Also, in general, our technique of pushing a pseudochord governed by the non-interfering invariant, might be useful in solving other problems. It is a way to navigate through a polygonal environment to a dead-end (nonredundant component). Perhaps it can be modified to navigate more intelligently, so that it can explore a general polygon, or find something more specific than just any nonredundant component.

Bibliography

[1] Nancy M. Amato, Michael T. Goodrich, and Edgar A. Ramos. Linear-time triangulation of a simple polygon made easier via randomization. In *Symposium on Computational Geometry*, pages 201–212, 2000.

[2] David Avis and Godfried T. Toussaint. An optimal algorithm for determining the visibility of a polygon from an edge. *IEEE Trans. Computers*, 30(12):910–914, 1981.

[3] Stephen Bahun and Anna Lubiw. Optimal schedules for 2-guard room search. In *CCCG*, pages 245–248, 2007.

[4] Binay K. Bhattacharya, Subir Kumar Ghosh, and Thomas C. Shermer. A linear time algorithm to remove winding of a simple polygon. *Comput. Geom.*, 33(3):165–173, 2006.

[5] Binay K. Bhattacharya and Asish Mukhopadhyay. Computing in linear time a chord from which a simple polygon is weakly internally visible. In *ISAAC*, pages 22–31, 1995.

[6] Binay K. Bhattacharya, John Z. Zhang, Qiaosheng Shi, and Tsunehiko Kameda. An optimal solution to room search problem. In *CCCG*, 2006.

[7] Bernard Chazelle. Triangulating a simple polygon in linear time. *Discrete & Computational Geometry*, 6:485–524, 1991.

[8] Bernard Chazelle and Janet Incerpi. Triangulation and shape-complexity. *ACM Trans. Graph.*, 3(2):135–152, 1984.

[9] V Chvtal. A combinatorial theorem in plane geometry. *Journal of Combinatorial Theory Series B*, 18(1):39–41, 1975.

[10] Gautam Das, Paul J. Heffernan, and Giri Narasimhan. Finding all weakly-visible chords of a polygon in linear time (extended abstract). In *SWAT*, pages 119–130, 1994.

[11] Gautam Das, Paul J. Heffernan, and Giri Narasimhan. Lr-visibility in polygons. *Comput. Geom.*, 7:37–57, 1997.

[12] Moshe Dror, Alon Efrat, Anna Lubiw, and Joseph S. B. Mitchell. Touring a sequence of polygons. In *In Proc. 35th Annu. ACM Sympos. Theory Comput*, pages 473–482. ACM Press, 2003.

[13] Hossam A. ElGindy and Godfried T. Toussaint. On geodesic properties of polygons relevant to linear time triangulation. *The Visual Computer*, 5(1&2):68–74, 1989.

[14] Steve Fisk. A short proof of chvátal's watchman theorem. *J. Comb. Theory, Ser. B*, 24(3):374, 1978.

[15] Alain Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. Graph.*, 3(2):153–174, 1984.

[16] Subir Kumar Ghosh. *Visibility Algorithms in the Plane*. Cambridge University Press, 2007.

[17] Leonidas J. Guibas, John Hershberger, Daniel Leven, Micha Sharir, and Robert Endre Tarjan. Linear-time algorithms for visibility and shortest path problems inside triangulated simple polygons. *Algorithmica*, 2:209–233, 1987.

[18] Mikael Hammar and Bengt J. Nilsson. Concerning the time bounds of existing shortest watchman route algorithms. In *In Proc. 11th International Symposium on Fundamentals of Computation Theory*, pages 210–221. Springer Verlag, 1997.

[19] Paul J. Heffernan. An optimal algorithm for the two-guard problem. *Int. J. Comput. Geometry Appl.*, 6(1):15–44, 1996.

[20] Tsunehiko Kameda and John Z. Zhang. Finding all door locations that make a room searchable. *Int. J. Comput. Geometry Appl.*, 20(2):175–201, 2010.

[21] D. T. Lee and Arthur K. Lin. Computational complexity of art gallery problems. *IEEE Transactions on Information Theory*, 32(2):276–282, 1986.

[22] Fajie Li and Reinhard Klette. Watchman route in a simple polygon with a rubberband algorithm. In *CCCG*, pages 1–4, 2010.

[23] J. O'Rourke. *Art gallery theorems and algorithms*. International series of monographs on computer science. Oxford University Press, 1987.

[24] Wei pang Chin and Simeon C. Ntafos. Optimum watchman routes. In *Symposium on Computational Geometry*, pages 24–33, 1986.

[25] Wei pang Chin and Simeon C. Ntafos. Shortest watchman routes in simple polygons. *Discrete & Computational Geometry*, 6:9–31, 1991.

[26] Jörg-Rüdiger Sack and Subhash Suri. An optimal algorithm for detecting weak visibility of a polygon. *IEEE Trans. Computers*, 39(10):1213–1219, 1990.

[27] T.C. Shermer. Recent results in art galleries [geometry]. *Proceedings of the IEEE*, 80(9):1384 –1399, sep 1992.

[28] Xuehou Tan. Searching a polygonal region by two guards. In *TAMC*, pages 262–273, 2007.

[29] Xuehou Tan. A unified and efficient solution to the room search problem. *Comput. Geom.*, 40(1):45–60, 2008.

[30] Xuehou Tan, Tomio Hirata, and Yasuyoshi Inagaki. An incremental algorithm for constructing shortest watchman routes. In *ISA*, pages 163–175, 1991.

[31] Xuehou Tan and Bo Jiang. Searching a polygonal region by two guards. *J. Comput. Sci. Technol.*, 23(5):728–739, 2008.

[32] Xuehou Tan and Bo Jiang. Optimum sweeps of simple polygons with two guards. In *FAW*, pages 304–315, 2010.

[33] John Z. Zhang. The two-guard polygon walk problem. In *TAMC*, pages 450–459, 2009.

[34] Zhong Zhang. *Applications of Visibility Space in Polygon Search Problems*. PhD thesis, Simon Fraser University, 2005.

[35] Junqiang Zhou and Simeon C. Ntafos. Two-guard art gallery problem. In *CCCG*, 2006.