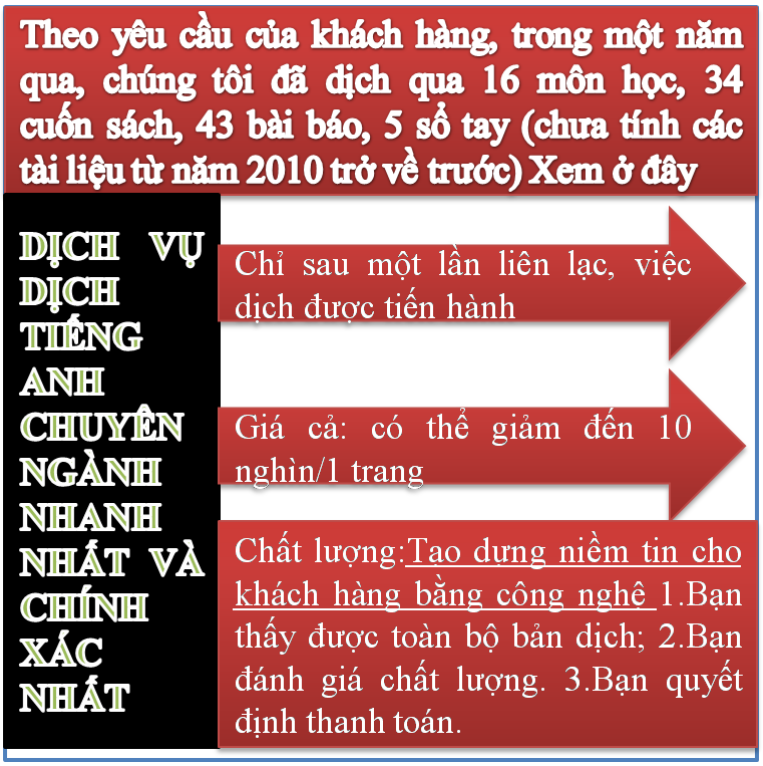


Tài liệu này được dịch sang tiếng việt bởi:

www.mientayvn.com

Tìm bản gốc tại thư mục này (copy link và dán hoặc nhấn Ctrl+Click):

https://drive.google.com/folderview?id=0B4rAPqlxIMRDSFE2RXQ2N3FtdDA&usp=sharing

Liên hệ để mua:

thanhlam1910_2006@yahoo.com hoặc frbwrthes@gmail.com hoặc số 0168 8557 403 (gặp Lâm)

Giá tiền: 1 nghìn /trang đơn (trang không chia cột); 500 VND/trang song ngữ

Dịch tài liệu của bạn: http://www.mientayvn.com/dich_tieng_anh_chuyen_nghanh.html

Computational Geometry Algorithms and Applications Third Edition	Hình học tính toán Các thuật toán và ứng dụng Tái bản lần III	
1.Computational Geometry Introduction Imagine you are walking on the campus of a university and suddenly you realize you have to make an urgent phone call. There are many public phones on campus and of course you want to go to the nearest one. But which one is the nearest? It would be helpful to have a map on which you could look up the nearest public phone, wherever on campus you are. The map should show a subdivision of the campus into regions, and for each region indicate the nearest public phone. What would these regions look like? And how could we compute them? Even though this is not such a terribly important issue, it describes the basics of a fundamental geometric concept, which plays a role in many applications. The subdivision of the campus is a so-called Voronoi diagram, and it will be studied in Chapter 7 in this book. It can be used to model trading areas of different cities, to	1.Nhập môn hình học tính toán Hãy tưởng tượng bạn đang đi bộ trong khuôn viên trường đại học và bỗng nhiên bạn thấy cần phải thực hiện một cuộc gọi khẩn cấp. Có nhiều trạm điện thoại công cộng trong khuôn viên trường và tất nhiên bạn muốn đến trạm nào gần nhất. Nhưng trạm nào gần nhất đây? Vì vậy, sẽ rất hữu ích nếu bạn có một bản đồ để tìm trạm điện thoại công cộng gần nhất khi bạn đi trong khuôn viên trường. Bản đồ sẽ biểu diễn các khu vực trong khuôn viên, và trạm điện thoại công cộng gần nhất trong mỗi khu vực đó. Những khu vực này có dạng như thế nào? Và làm sao người ta có thể tính toán được chúng? Mặc dù đây không phải là một vấn đề quá quan trọng, nhưng nó giúp chúng ta hình dung được một khái niệm cơ bản của hình học, một khái niệm đóng vai trò quan trọng trong nhiều ứng dụng. Việc chia nhỏ khuôn viên được gọi là sơ đồ Voronoi, và chúng ta sẽ nghiên cứu khái niệm đó trong chương 7 của sách này. Nó có thể được sử dụng	

guide robots, and even to describe and simulate the growth of crystals. Computing a geometric structure like a Voronoi diagram requires geometric algorithms. Such algorithms form the topic of this book.	để mô hình hóa các khu vực thương mại của thành phố, dẫn đường cho robot, và thậm chí để mô tả và mô phỏng sự phát triển của các tinh thể. Tính toán một cấu trúc hình học như sơ đồ Voronoi đòi hỏi các thuật toán hình học. Các thuật toán như thế cũng là chủ đề của sách này.	
A second example. Assume you located the closest public phone. With a campus map in hand you will probably have little problem in getting to the phone along a reasonably short path, without hitting walls and other objects. But programming a robot to perform the same task is a lot more difficult. Again, the heart of the problem is geometric: given a collection of geometric obstacles, we have to find a short connection between two points, avoiding collisions with the obstacles. Solving this so-called motion planning problem is of crucial importance in robotics. Chapters 13 and 15 deal with geometric algorithms required for motion planning.	Ví dụ thứ hai. Giả sử bạn đã xác định được trạm điện thoại công cộng gần nhất. Khi có bản đồ trong tay, bạn có thể dễ dàng đi đến trạm điện thoại đó theo đường ngắn nhất mà không chạm tường hoặc các vật cản khác. Nhưng lập trình để robot thực hiện một nhiệm vụ tương tự lại khó khăn hơn nhiều. Một lần nữa, trọng tâm của vấn đề lại là hình học: cho một tập hợp các chướng ngại vật hình học, chúng ta phải tìm đường đi ngắn cho vật giữa hai điểm, sao cho nó không chạm phải các chướng ngại vật. Đây được gọi là bài toán hoạch định chuyển động cực kỳ quan trọng trong lĩnh vực robot học. Chương 13 và 15 sẽ đề cập đến các thuật toán hình học cần thiết để hoạch định chuyển động.	
A third example. Assume you don’t have one map but two: one with a description of the various buildings, including the public phones, and one	Ví dụ thứ ba. Giả sử bạn không có một bản đồ mà có hai: Một bản đồ mô tả các tòa nhà khác nhau, kể cả	

indicating the roads on the campus. To plan a motion to the public phone we have to overlay these maps, that is, we have to combine the information in the two maps. Overlaying maps is one of the basic operations of geographic information systems. It involves locating the position of objects from one map in the other, computing the intersection of various features, and so on. Chapter 2 deals with this problem. These are just three examples of geometric problems requiring carefully de-signed geometric algorithms for their solution. In the 1970s the field of computational geometry emerged, dealing with such geometric problems. It can be defined as the systematic study of algorithms and data structures for geometric objects, with a focus on exact algorithms that are asymptotically fast. Many researchers were attracted by the challenges posed by the geometric problems. The road from problem formulation to efficient and elegant solutions has often been long, with many difficult and sub-optimal intermediate results. Today there is a rich	các trạm điện thoại công cộng, và một bản đồ chỉ dẫn đường trong khuôn viên. Để xác định được đường đến trạm điện thoại công cộng, chúng ta phải chồng xếp hai bản đồ này, có nghĩa là, chúng ta phải kết hợp thông tin trong hai bản đồ. Chồng xếp bản đồ là một trong những thuật toán cơ bản của các hệ thống thông tin địa lý. Nó liên quan đến việc định vị các đối tượng từ một bản đồ trong bản đồ kia, tính toán sự giao nhau của các thuộc tính khác nhau, và v.v.… Chương 2 sẽ đề cập đến vấn đề này. Đây chỉ là ba ví dụ về các vấn đề hình học đòi hỏi các thuật toán hình học được thiết kế cẩn thận để giải quyết chúng. Vào những năm 1970, lĩnh vực hình học tính toán đã nổi lên, nghiên cứu các vấn đề hình học như thế. Chúng ta có thể định nghĩa nó là một nghiên cứu có hệ thống về các thuật toán và các cấu trúc dữ liệu cho các đối tượng hình học, chủ yếu tập trung vào các thuật toán chính xác tiệm cận nhanh. Nhiều nhà nghiên cứu bị cuốn hút bởi những thách thức xuất hiện từ các bài toán hình học. Con đường từ phát biểu vấn đề tới các giải pháp hiệu quả và tinh tế thường là một quá trình lâu dài, với nhiều khó khăn và các kết quả	

collection of geometric algorithms that are efficient, and relatively easy to understand and implement. This book describes the most important notions, techniques, algorithms, and data structures from computational geometry in a way that we hope will be attractive to readers who are interested in applying results from computational geometry. Each chapter is motivated with a real computational problem that requires geometric algorithms for its solution. To show the wide applicability of computational geometry, the problems were taken from various application areas: robotics, computer graphics, CAD/CAM, and geographic information systems. You should not expect ready-to-implement software solutions for major problems in the application areas. Every chapter deals with a single concept in computational geometry; the applications only serve to introduce and motivate the concepts. They also illustrate the process of modeling an engineering problem and finding an exact solution.	trung gian chưa tối ưu. Ngày nay, đã có một lượng dồi dào các thuật toán hình học hiệu quả và tương đối dễ hiểu cũng như dễ thực thi. Cuốn sách này mô tả các khái niệm, các kỹ thuật, các thuật toán và cấu trúc dữ liệu quan trọng nhất trong hình học tính toán nhắm đến những độc giả muốn áp dụng các kết quả của hình học tính toán để giải quyết các vấn đề cụ thể. Mỗi chương được bắt đầu bằng một bài toán tính toán thực tế, và những bài toán này cần các thuật toán hình học để giải quyết. Để giúp người đọc thấy được ứng dụng rộng rãi của hình học tính toán, chúng tôi lấy các ví dụ từ nhiều lĩnh vực ứng dụng khác nhau: robot, đồ họa máy tính, CAD / CAM, và các hệ thống thông tin địa lý. Bạn không nên trong chờ các giải pháp phần mềm dễ thực thi cho các bài toán lớn trong các lĩnh vực ứng dụng. Mỗi chương sẽ đề cập đến một khái niệm duy nhất trong hình học tính toán; ứng dụng chỉ đóng vai trò giới thiệu và thúc đẩy việc tìm hiểu khái niệm. Chúng cũng minh họa quá trình mô hình hóa một bài toán kỹ thuật và tìm một nghiệm chính xác.	

1.1 An Example: Convex Hulls Good solutions to algorithmic problems of a geometric nature are mostly based on two ingredients. One is a thorough understanding of the geometric properties of the problem, the other is a proper application of algorithmic techniques and data structures. If you don't understand the geometry of the problem, all the algorithms of the world won't help you to solve it efficiently. On the other hand, even if you perfectly understand the geometry of the problem, it is hard to solve it effectively if you don't know the right algorithmic techniques. This book will give you a thorough understanding of the most important geometric concepts and algorithmic techniques. To illustrate the issues that arise in developing a geometric algorithm, this section deals with one of the first problems that was studied in computational geometry: the computation of planar convex hulls. We'll skip the motivation for this problem here; if you are interested you can read the introduction to Chapter 11, where we study convex hulls	1.1 Ví dụ: Các bao lồi Giải pháp tốt cho các bài toán giải thuật có bản chất hình học dựa trên hai yếu tố. Một là hiểu thấu đáo về các tính chất hình học của bài toán, và thứ hai là áp dụng các kỹ thuật giải thuật và cấu trúc dữ liệu thích hợp. Nếu bạn không hiểu tính chất hình học của một bài toán, tất cả các giải thuật trong thế giới này sẽ không giúp bạn giải quyết nó hiệu quả. Mặt khác, ngay cả khi bạn đã hiểu thấu đáo các tính chất hình học của bài toán, sẽ rất khó để giải nó hiệu quả nếu bạn không áp dụng các kỹ thuật giải thuật thích hợp. Sách này sẽ giúp bạn hiểu rõ các khái niệm hình học quan trọng nhất và các kỹ thuật giải thuật. Để minh họa các vấn đề nảy sinh trong việc xây dựng thuật toán hình học, phần này sẽ xét một trong những bài toán đầu tiên được nghiên cứu trong hình học tính toán: tính toán các bao lồi phẳng. Chúng tôi sẽ bỏ qua nguồn gốc của bài toán này, nếu bạn quan tâm, bạn có thể đọc phần giới thiệu ở chương 11, trong chương đó chúng tôi xét các bao lồi trong không gian	

in 3-dimensional space. A subset S of the plane is called convex if and only if for any pair of points p, q e S the line segment pq is completely contained in S. The convex hull CH(S) of a set S is the smallest convex set that contains S. To be more precise, it is the intersection of all convex sets that contain S. We will study the problem of computing the convex hull of a finite set P of n points in the plane. We can visualize what the convex hull looks like by a thought experiment. Imagine that the points are nails sticking out of the plane, take an elastic rubber band, hold it around the nails, and let it go. It will snap around the nails, minimizing its length. The area enclosed by the rubber band is the convex hull of P. This leads to an alternative definition of the convex hull of a finite set P of points in the plane: it is the unique convex polygon whose vertices are points from P and that contains all points of P. Of course we should prove rigorously that this is well defined—that is, that the polygon is unique—and that the definition is equivalent to the one given	3 chiều. Một tập hợp con S của mặt phẳng được gọi là lồi khi và chỉ khi đối với bất kỳ cặp điểm p,q ∈ S, đoạn thẳng $\overline{pq}$ được chứa hoàn toàn trong S. Bao lồi CH(S) của một tập S là tập lồi nhỏ nhất chứa S. Nói chính xác hơn, nó là giao của tất cả các tập lồi chứa S. Chúng ta sẽ nghiên cứu bài toán xác định bao lồi của một tập hợp hữu hạn P chứa n điểm trong mặt phẳng. Chúng ta có thể hình dung bao lồi có dạng như thế nào qua một thí nghiệm tưởng tượng như sau. Giả sử các điểm là các móng nhô ra khỏi mặt phẳng, lấy một dây cao su đàn hồi, giữ nó xung quanh các móng, và cứ để nó co lại tự nhiên. Nó sẽ bao xung quanh các móng tay, làm chiều dài của nó cực tiểu. Vùng được bao quanh bởi dây cao su chính là bao lồi của P. Điều này dẫn đến một định nghĩa khác về bao lồi của một tập hữu hạn P của các điểm trong mặt phẳng: nó là đa giác lồi duy nhất có các đỉnh là các điểm từ P và chứa tất cả các điểm của P. Dĩ nhiên chúng ta có thể chứng minh một cách chặt chẽ rằng điều này tồn tại, tức là, đa giác là duy nhất và định nghĩa tương	

earlier, but let's skip that in this introductory chapter.	đương với định nghĩa đã được đưa ra trước đây, nhưng chúng ta sẽ bỏ qua điều đó trong chương nhập môn này.	
How do we compute the convex hull? Before we can answer this question we must ask another question: what does it mean to compute the convex hull? As we have seen, the convex hull of P is a convex polygon. A natural way to represent a polygon is by listing its vertices in clockwise order, starting with an arbitrary one. So the problem we want to solve is this: given a set P = {pi, pn} of points in the plane, compute a list that contains those points from P that are the vertices of CH(P), listed in clockwise order.	Vậy chúng ta tính toán bao lồi như thế nào? Trước khi trả lời câu hỏi này chúng ta phải trả lời câu hỏi khác: Tính toán bao lồi có ngĩa là gì? Như chúng ta đã thấy, bao lồi của P là một đa giác lồi. Một cách tự nhiên để biểu diễn đa giác là liệt kê các đỉnh của nó theo thứ tự cùng chiều kim đồng hồ, bắt đầu từ một đỉnh tùy ý. Vậy vấn đề chúng ta cần giải quyết là như thế này: cho một tập hợp P = {pi, pn} của các điểm trong mặt phẳng, tìm một danh sách chứa những điểm đó từ P, với điều kiện chúng là các đỉnh của CH (P), được liệt kê theo chiều kim đồng hồ.	
input = set of points: Pi,P2,P3,P4,P5,P6,P7,P8,P9	đầu vào = tập hợp các điểm: Pi, P2, P3, P4, P5, P6, P7, P8, P9	
output = representation of the convex hull: P4, P5, P8, P2, P9	Đầu ra =biểu diễn bao lồi: P4, P5, P8, P2, P9	
The first definition of convex hulls is of little help when we want to design an algorithm to compute the convex hull. It talks about the intersection of all convex sets containing P, of which there are infinitely many. The observation that	Định nghĩa đầu tiên về bao lồi có vẻ chưa hữu dụng trong trường hợp chúng ta muốn thiết kế thuật toán để tính bao lồi. Nó đề cập đến giao của tất cả các tập lồi chứa P, trong số đó có nhiều tập vô hạn. Tính	

CH(P) is a convex polygon is more useful. Let's see what the edges of CH(P) are. Both endpoints P and q of such an edge are points of P, and if we direct the line through P and q such that CH(P) lies to the right, then all the points of P must lie to the right of this line. The reverse is also true: if all points of P \{P, q} lie to the right of the directed line through P and q, then pq is an edge of CH(P). Now that we understand the geometry of the problem a little bit better we can develop an algorithm. We will describe it in a style of pseudocode we will use throughout this book. Algorithm SLOWCONVEXHULL(P) Input. A set P of points in the plane. Output. A list L containing the vertices of CH(P) in clockwise order. 1.E ^ 0. 2.for all ordered pairs (p, q) e P X P with p not equal to q 3.do valid <— true 4.for all points r e P not equal to p or q	chất CH (P) là một đa giác lồi hữu ích hơn. Chúng ta hãy xét các cạnh của CH (P) là gì. Cả hai điểm mút p và q của một cạnh như thế là những điểm thuộc P, và nếu chúng ta hướng đường thẳng qua p và q sao cho CH (P) nằm bên phải, thì tất cả các điểm của P phải nằm bên phải đường này. Điều ngược lại cũng đúng: nếu tất cả các điểm thuộc P \ {P, q} nằm ở bên phải của đường có hướng qua p và q, thì pq là một cạnh của CH (P). Bây giờ chúng ta đã hiểu được hình học của bài toán tốt hơn một chút và chúng ta có thể xây dựng một thuật toán. Chúng tôi sẽ mô tả nó theo kiểu mã giả và sẽ làm như thế trong toàn bộ sách này. Thuật toán SLOWCONVEXHULL (P) Đầu vào. Một tập hợp P bao gồm các điểm trong mặt phẳng. Đầu ra. Một danh sách L chứa các đỉnh của CH (P) theo chiều kim đồng hồ. 1. E ^ 0. 2. for tất cả các cặp có thứ tự (p, q) e PXP với p không bằng q 3. do hợp lệ <- true	

5.do if r lies to the left of the directed line from p to q 6.then valid ^ false. 7.if valid then Add the directed edge pq to E. 8. From the set E of edges construct a list L of vertices of CH(P), sorted in clockwise order. Two steps in the algorithm are perhaps not entirely clear. The first one is line 5: how do we test whether a point lies to the left or to the right of a directed line? This is one of the primitive operations required in most geometric algorithms. Throughout this book we assume that such operations are available. It is clear that they can be performed in constant time so the actual implementation will not affect the asymptotic running time in order of magnitude. This is not to say that such primitive operations are unimportant or trivial. They are not easy to implement correctly and their implementation will affect the actual running time of the algorithm. Fortunately,	4.for tất cả các điểm …. không bằng p hoặc q 5. do nếu r nằm bên trái của đường có hướng từ p đến q 6. then hợp lệ---false. 7. if hợp lệ then Thêm cạnh có hướng pq vào E. 8. Từ tập hợp E của các cạnh xây dựng một danh sách L các đỉnh của CH (P), được sắp xếp theo chiều kim đồng hồ. Có lẽ còn hai bước trong thuật toán chưa hoàn toàn rõ ràng. Bước đầu tiên ở dòng thứ 5: làm sao để chúng ta kiểm tra một điểm nằm bên trái hoặc bên phải của đường có hướng? Đây là một trong những phép toán cơ bản cần thiết trong đa số các thuật toán hình học. Trong toàn bộ sách này, chúng tôi giả sử rằng các phép toán như thế đã có sẵn. Rõ ràng, chúng có thể được thực hiện trong khoảng thời gian hằng, vì vậy quá trình thực thi thực tế sẽ không ảnh hưởng đến thời gian chạy tiệm cận vào bậc độ lớn. Điều này không có nghĩa là các thuật toán cơ bản như thế là không quan trọng hoặc tầm thường. Chúng không dễ thực thi chính xác và việc thực thi chúng sẽ ảnh	

software libraries containing such primitive operations are nowadays available. We conclude that we don't have to worry about the test in line 5; we may assume that we have a function available performing the test for us in constant time.	hưởng đến thời gian chạy thực tế của thuật toán. Tuy nhiên, hiện nay đã có những thư viện phần mềm chứa những phép toán cơ bản như thế. Chúng tôi dám chắc rằng bạn không cần phải lo ngại về phép kiểm tra ở dòng số 5, chúng ta có thể giả sử rằng chúng ta có sẵn một hàm thực hiện kiểm tra cho chúng ta trong khoảng thời gian hằng.	
The other step of the algorithm that requires some explanation is the last one. In the loop of lines 2-7 we determine the set E of convex hull edges. From E we can construct the list L as follows. The edges in E are directed, so we can speak about the origin and the destination of an edge. Because the edges are directed such that the other points lie to their right, the destination of an edge comes after its origin when the vertices are listed in clockwise order. Now remove an arbitrary edge e1 from E. Put the origin of e1 as the first point into L, and the destination as the second point. Find the edge e2 in E whose origin is the destination of e1, remove it from E, and append its destination to L. Next, find the edge e3 whose origin is the destination of e2, remove	Một bước khác trong thuật toán cũng cần giải thích thêm là bước cuối cùng. Trong vòng lặp từ dòng 2-7 chúng tôi xác định tập E các cạnh bao lồi. Từ E, chúng ta có thể xây dựng một danh sách L như sau. Các cạnh trong E có hướng, vì vậy chúng ta có thể nói về gốc và ngọn của một cạnh. Bởi vì các cạnh có hướng sao cho các điểm khác nằm bên phải của chúng, ngọn của một cạnh sẽ đến sau gốc của nó khi các đỉnh được liệt kê theo chiều kim đồng hồ. Bây giờ loại bỏ một cạnh e1 bất kỳ khỏi E. Đặt gốc của e1 như điểm đầu tiên vào L, và ngọn là điểm thứ hai. Tìm cạnh e2 trong E sao cho gốc của nó là ngọn của e1, loại bỏ nó khỏi E, và nối ngọn của nó đến L. Tiếp theo, tìm cạnh e3 sao cho gốc của nó là ngọn của e2, loại bỏ nó	

it from E, and append its destination to L. We continue in this manner until there is only one edge left in E. Then we are done; the destination of the remaining edge is necessarily the origin of e1, which is already the first point in L. A simple implementation of this procedure takes O(n2) time. This can easily be improved to O(nlogn), but the time required for the rest of the algorithm dominates the total running time anyway. Analyzing the time complexity of SlowConvexHull is easy. We check n2 - n pairs of points. For each pair we look at the n - 2 other points to see whether they all lie on the right side. This will take O(n3) time in total. The final step takes O(n2) time, so the total running time is O(n3). An algorithm with a cubic running time is too slow to be of practical use for anything but the smallest input sets. The problem is that we did not use any clever algorithmic design techniques; we just translated the geometric insight into an algorithm in a brute-force manner. But before we try to do better, it is useful to make several observations about	khỏi E, và nối ngọn của nó đến L. Chúng ta tiếp tục quy trình này cho đến khi chỉ còn một cạnh còn lại trong E. Sau đó, chúng ta hoàn thành, ngọn của cạnh còn lại bắt buộc là gốc của e1, đó chính là điểm đầu tiên trong L. Việc thực thi quá trình đơn giản này mất khoảng thời gian là O (n2). Có thể cải thiện thêm để đạt được thời gian O (nlogn), nhưng dù sao đi nữa, thời gian cần thiết cho phần còn lại của giải thuật mới chiếm phần lớn tổng thời gian chạy. Việc phân tích độ phức tạp thời gian của thuật toán SlowConvexHull khá dễ. Chúng ta kiểm tra n^2 - n cặp điểm. Đối với mỗi cặp, chúng ta xét n - 2 điểm khác để xem liệu tất cả chúng có nằm phía bên phải hay không. Tổng thời gian cho quá trình này là O(n3). Bước cuối cùng mất khoảng thời gian là O (n2), vì vậy tổng thời gian chạy là O (n3). Một thuật toán có thời gian chạy mũ ba là quá chậm để ứng dụng thực tế nhưng các tập hợp đầu vào nhỏ nhất. Vấn đề là chúng ta đã không sử dụng bất kỳ kỹ thuật thiết kế thuật toán khéo léo nào, chúng ta chỉ chuyển hiểu biết hình học vào trong một thuật toán theo kiểu vét cạn. Nhưng trước khi chúng ta cố gắng cải	

this algorithm.	thiện điều này, chúng ta nên nhìn lại thuật toán này.	
We have been a bit careless when deriving the criterion of when a pair p, q defines an edge of CH(P). A point r does not always lie to the right or to the left of the line through p and q, it can also happen that it lies on this line. What should we do then? This is what we call a degenerate case, or a degeneracy for short. We prefer to ignore such situations when we first think about a problem, so that we don't get confused when we try to figure out the geometric properties of a problem. However, these situations do arise in practice. For instance, if we create the points by clicking on a screen with a mouse, all points will have small integer coordinates, and it is quite likely that we will create three points on a line.	Chúng ta hơi bất cẩn khi rút ra tiêu chuẩn khi nào cặp p, q xác định một cạnh của CH (P). Một điểm r không phải luôn luôn lúc nào cũng nằm bên phải hoặc bên trái của đường thẳng đi qua p và q, cũng có khi nó ngẫu nhiên nằm trên đường này. Thế thì chúng ta nên làm gì? Chúng ta gọi đây là trường hợp suy biến, hoặc nói ngắn gọn là suy biến. Chúng tôi muốn bỏ qua các tình huống như vậy khi lần đầu tiên chúng ta xét bài toán, vì vậy chúng ta không bị lẫn lộn về thời điểm rút ra các tính chất hình học của bài toán. Tuy nhiên, những tình huống như thế cũng có thể nảy sinh trong thực tế. Ví dụ, nếu chúng ta tạo ra các điểm bằng cách nhấp chuột trên màn hình, tất cả các điểm có tọa độ nguyên nhỏ, và rất có khả năng chúng ta sẽ tạo ra ba điểm trên một đường thẳng.	
To make the algorithm correct in the presence of degeneracies we must reformulate the criterion above as follows: a directed edge pq is an edge of CH(P) if and only if all other points r e P lie either strictly to the right of the directed line through p and q, or they lie	Để thực hiện các thuật toán chính xác trong trường hợp suy biến, chúng ta phải phát biểu lại tiêu chuẩn trên như sau: một cạnh pq có hướng là cạnh của CH (P) khi và chỉ khi tất cả các điểm khác……nằm ở bên phải của đường có hướng qua p và q, hoặc	

on the open line segment pq. (We assume that there are no coinciding points in p.) So we have to replace line 5 of the algorithm by this more complicated test. We have been ignoring another important issue that can influence the correctness of the result of our algorithm. We implicitly assumed that we can somehow test exactly whether a point lies to the right or to the left of a given line. This is not necessarily true: if the points are given in floating point coordinates and the computations are done using floating point arithmetic, then there will be rounding errors that may distort the outcome of tests. Imagine that there are three points p, q, and r, that are nearly collinear, and that all other points lie far to the right of them. Our algorithm tests the pairs (p, q), (r, q), and (p, r). Since these points are nearly collinear, it is possible that the rounding errors lead us to decide that r lies to the right of the line from p to q, that p lies to the right of the line from r to q, and that q lies to the right of the line from p to r. Of course this is geometrically impossible—but the floating point	chúng nằm trên đoạn thẳng mở pq. (Chúng ta giả sử rằng không có các điểm trùng trong p.) Vì vậy, chúng ta phải thay thế dòng thứ 5 của thuật toán bằng phép kiểm tra phức tạp này. Chúng ta đã bỏ qua một vấn đề quan trọng khác có thể ảnh hưởng đến tính chính xác của các kết quả thuật toán. Chúng ta đã ngầm giả định rằng, bằng cách nào đó, chúng ta có thể kiểm tra chính xác một điểm nằm bên phải hay bên trái của một đường nhất định. Điều này không nhất thiết là đúng: nếu các điểm được cho trong tọa độ dấu chấm động, thì sẽ xuất hiện sai số làm tròn có thể làm sai lệch các kết quả của phép kiểm tra. Hãy tưởng tượng rằng có ba điểm p, q, r, gần như thẳng hàng, và tất cả các điểm khác nằm xa bên phải chúng. Thuật toán của chúng ta kiểm tra các cặp (p, q), (r, q), và (p, r). Vì các điểm này gần thẳng hàng, có khả năng là sai số làm tròn làm chúng ta kết luận rằng r nằm bên phải của đường thẳng đi từ p đến q, và p nằm bên phải của đường thẳng từ r đến q, và q nằm bên phải của đường thẳng từ p đến r. Tất nhiên, về mặt hình học, điều này không thể xảy ra, nhưng số học dấu chấm	

arithmetic doesn't know that! In this case the algorithm will accept all three edges. Even worse, all three tests could give the opposite answer, in which case the algorithm rejects all three edges, leading to a gap in the boundary of the convex hull. And this leads to a serious problem when we try to construct the sorted list of convex hull vertices in the last step of our algorithm. This step assumes that there is exactly one edge starting in every convex hull vertex, and exactly one edge ending there. Due to the rounding errors there can suddenly be two, or no, edges starting in vertex p. This can cause the program implementing our simple algorithm to crash, since the last step has not been designed to deal with such inconsistent data. Although we have proven the algorithm to be correct and to handle all special cases, it is not robust: small errors in the computations can make it fail in completely unexpected ways. The problem is that we have proven the correctness assuming that we can compute exactly with real numbers.	động không biết điều đó! Trong trường hợp này, thuật toán sẽ chấp nhận tất cả ba cạnh. Thậm chí, tệ hơn nữa, tất cả ba phép kiểm tra cho ra kết quả trái ngược, trong trường hợp này, thuật toán loại bỏ cả ba cạnh, dẫn đến một khoảng trống ở biên của bao lồi. Và điều này dẫn đến một vấn đề nghiêm trọng khi chúng ta xây dựng danh sách các đỉnh được sắp xếp của bao lồi ở bước cuối cùng của thuật toán của chúng ta. Bước này giả sử rằng có đúng một cạnh bắt đầu ở mỗi đỉnh của bao lồi, và có đúng một cạnh kết thúc ở đó. Do các sai số làm tròn, có thể đột nhiên có hai, hoặc không có cạnh nào bắt đầu ở đỉnh p. Điều này có thể làm sụp đổ việc thực thi thuật toán đơn giản của chúng ta, bởi vì khi thiết kế bước cuối cùng chúng ta chưa tính đến các dữ liệu không nhất quán như thế. Mặc dù chúng ta đã chứng minh thuật toán chính xác và có thể xử lý được tất cả các trường hợp đặc biệt, nhưng nó không mạnh mẽ: các sai số nhỏ trong tính toán có thể làm cho nó sai theo những cách chúng ta không lường trước được. Vấn đề là chúng ta đã chứng minh tính chính xác giả sử rằng chúng ta có	

We have designed our first geometric algorithm. It computes the convex hull of a set of points in the plane. However, it is quite slow—its running time is O(n3)—, it deals with degenerate cases in an awkward way, and it is not robust. We should try to do better.	thể tính chính xác với các số thực. Chúng ta vừa hoàn thành việc thiết kế thuật toán hình học đầu tiên. Nó tính bao lồi của một tập hợp điểm trong mặt phẳng. Tuy nhiên, nó quá chậm-thời gian chạy của nó là O (N3) -, nó có thể giải các trường hợp suy biến nhưng rất khó khăn, và nó không mạnh. Chúng ta nên cố gắng cải thiện nó.	
To this end we apply a standard algorithmic design technique: we will develop an incremental algorithm. This means that we will add the points in P one by one, updating our solution after each addition. We give this incremental approach a geometric flavor by adding the points from left to right. So we first sort the points by x-coordinate, obtaining a sorted sequence p1pn, and then we add them in that order. Because we are working from left to right, it would be convenient if the convex hull vertices were also ordered from left to right as they occur along the boundary. But this is not the case. Therefore we first compute only those convex hull vertices that lie on the upper hull, which is the part of the convex hull running	Trong phần cuối này, chúng ta áp dụng một kỹ thuật thiết kế thuật toán tiêu chuẩn: chúng ta sẽ xây dựng một thuật toán gia tăng. Điều này có nghĩa là chúng ta sẽ thêm từng điểm một vào P, cập nhật nghiệm của chúng ta sau mỗi lần thêm vào. Chúng ta sẽ cho giải thuật gia tăng này một hương vị hình học bằng cách thêm vào các điểm từ trái sang phải. Vì vậy, trước hết chúng ta sắp xếp các điểm theo tọa độ x, thu được một chuỗi được sắp xếp p1pn, và sau đó chúng ta thêm chúng theo thứ tự đó. Bởi vì chúng ta đang tiến hành từ trái sang phải, sẽ tiện lợi hơn nếu các đỉnh của bao lồi được xếp thứ tự từ trái sang phải khi chúng xuất hiện dọc theo biên. Nhưng điều này không đúng. Vì vậy, trước hết, chúng ta chỉ tính toán các đỉnh bao lồi nằm trên bao trên, đó là	

from the leftmost point p1 to the rightmost point pn when the vertices are listed in clockwise order. In other words, the upper hull contains the convex hull edges bounding the convex hull from above. In a second scan, which is performed from right to left, we compute the remaining part of the convex hull, the lower hull. The basic step in the incremental algorithm is the update of the upper hull after adding a point pi. In other words, given the upper hull of the points p1;..., p—1, we have to compute the upper hull of p1;..., pi. This can be done as follows. When we walk around the boundary of a polygon in clockwise order, we make a turn at every vertex. For an arbitrary polygon this can be both a right turn and a left turn, but for a convex polygon every turn must be a right turn. This suggests handling the addition of pi in the following way. Let Lupper be a list that stores the upper vertices in left-to-right order. We first append pi to Lupper. This is correct because pi is the rightmost point of the ones added so far, so it must be on the upper hull. Next, we check whether the last three	phần bao lồi chạy từ điểm trái cùng p1 đến điểm phải cùng pn khi các đỉnh được liệt kê theo chiều kim đồng hồ. Nói cách khác, bao trên chứa các cạnh bao lồi bao quanh bao lồi từ trên. Ở lần quét thứ hai, chúng ta thực hiện từ phải sang trái, chúng ta tính toán phần còn lại của bao lồi, bao dưới. Bước cơ bản trong thuật toán gia tăng là cập nhật bao trên sau khi thêm một điểm pi. Nói cách khác, với một bao trên cho trước của các điểm p1, ..., p-1, chúng ta phải tính bao trên của p1, ..., pi. Điều này có thể được thực hiện như sau. Khi chúng ta đi quanh biên của đa giác theo chiều kim đồng hồ, chúng ta đã đi một vòng quanh mỗi đỉnh. Đối với một đa giác tùy ý, đây có thể là vòng trái hoặc vòng phải, nhưng đối với một đa giác lồi mỗi vòng đều là vòng phải. Điều này gợi ý cho chúng ta cách thức thêm pi như sau. Giả sử Lupper là một danh sách lưu trữ các đỉnh trên theo thứ tự trái sang phải. Trước tiên chúng ta thêm pi vào Lupper. Điều này là đúng bởi vì pi là điểm bên phải cùng của các điểm được cộng đến lúc này, vì vậy nó phải nằm ở bao trên. Tiếp theo, chúng ta kiểm	

points in Lupper make a right turn. If this is the case there is nothing more to do; Lupper contains the vertices of the upper hull of p1,..., pi, and we can proceed to the next point, pi+1. But if the last three points make a left turn, we have to delete the middle one from the upper hull. In this case we are not finished yet: it could be that the new last three points still do not make a right turn, in which case we again have to delete the middle one. We continue in this manner until the last three points make a right turn, or until there are only two points left. We now give the algorithm in pseudocode. The pseudocode computes both the upper hull and the lower hull. The latter is done by treating the points from right to left, analogous to the computation of the upper hull. AlgorithmCONVEXHULL(P) Input. A set P of points in the plane. Output. A list containing the vertices of CH(P) in clockwise order. 1. Sort the points by x-coordinate, resulting in a sequence p1,..., pn.	tra xem ba điểm cuối cùng nằm trong Lupper có tạo ra vòng phải hay không. Nếu điều này đúng, chúng ta không cần làm thêm điều gì nữa; Lupper chứa các đỉnh của bao trên của p1, ..., pi, và chúng ta có thể tiếp tục với điểm tiếp theo, pi+1. Nhưng nếu ba điểm cuối tạo ra một vòng trái, chúng ta phải xóa điểm ở giữa từ bao trên. Trong trường hợp này, chúng ta vẫn chưa hoàn thành: có thể ba điểm mới vẫn chưa tạo tạo ra một vòng phải, trong trường hợp này chúng ta phải xóa điểm ở giữa. Chúng ta tiếp tục theo cách này cho đến khi ba điểm cuối cùng tạo ra một vòng phải, hoặc cho đến khi chỉ còn hai điểm. Bây giờ chúng ta đưa ra thuật toán dưới dạng mã giả. Mã giả tính toán cả bao trên và bao dưới. Trường hợp sau được thực hiện bằng cách xét các điểm từ phải sang trái, tương tự như tính toán bao trên. Thuật toán CONVEXHULL (P) Đầu vào. Một tập hợp P các điểm trong mặt phẳng. Đầu ra. Một danh sách chứa các đỉnh của CH (P) theo chiều kim đồng hồ.	

2. Put the points p1 and p2 in a list Lupper, with p1 as the first point. 3. for i ^ 3 to n 4. do Append pi to Lupper. 5. while Lupper contains more than two points and the last three points in Lupper do not make a right turn 6. do Delete the middle of the last three points from Lupper. 7. Put the points pn and pn-1 in a list Liower, with pn as the first point. 8. for i ^ n - 2 downto i 9. do Append Pi to Llower. 10. while Llower contains more than 2 points and the last three points in Llower do not make a right turn 11. do Delete the middle of the last three points from Llower. 12. Remove the first and the last point from Llower to avoid duplication of the points where the upper and lower hull meet. 13. Append Llower to	1.Sắp xếp các điểm theo tọa độ x, cho ra chuỗi p1, ..., pn. 2. Đặt các điểm p1 và p2 trong danh sách Lupper, với p1 là điểm đầu tiên. 3. for ^ 3 to n 4. do Thêm pi vào Lupper. 5. While Lupper chứa nhiều hơn hai điểm và ba điểm cuối cùng trong Lupper không tạo ra vòng phải 6. do Xóa điểm giữa trong ba điểm cuối cùng từ Lupper. 7. Đặt các điểm pn và pn-1 vào danh sách Liower, với pn là điểm đầu tiên. 8. for i ^ n - 2 downto i 9. do Thêm Pi vào Llower. 10. while Llower chứa nhiều hơn 2 điểm và ba điểm cuối cùng trong Llower không tạo ra vòng phải 11. do Xóa điểm giữa trong ba điểm cuối từ Llower. 12. Loại bỏ điểm đầu tiên và điểm cuối cùng từ Llower để tránh trùng lặp trong những điểm bao trên và	

Lupper, and call the resulting list L. 14. return L Once again, when we look closer we realize that the above algorithm is not correct. Without mentioning it, we made the assumption that no two points have the same x-coordinate. If this assumption is not valid the order on x-coordinate is not well defined. Fortunately, this turns out not to be a serious problem. We only have to generalize the ordering in a suitable way: rather than using only the x-coordinate of the points to define the order, we use the lexicographic order. This means that we first sort by x-coordinate, and if points have the same x-coordinate we sort them by y-coordinate. Another special case we have ignored is that the three points for which we have to determine whether they make a left or a right turn lie on a straight line. In this case the middle point should not occur on the convex hull, so collinear points must be treated as if they make a left turn. In other words, we should use a test that returns true if the three points make a right turn, and false	dưới gặp nhau. 13. Thêm Llower vào Lupper, và gọi danh sách L cuối cùng. 14. trả về L Một lần nữa, khi chúng ta xét kỹ hơn, chúng ta thấy rằng thuật toán trên không chính xác. Chúng ta đã giả thiết rằng không có hai điểm nào có cùng tọa độ x. Nếu điều này không đúng, thứ tự trên tọa độ x không rõ ràng. Tuy nhiên, hóa ra điều này không phải là vấn đề nghiêm trọng. Chúng ta chỉ cần tổng quát hóa thứ tự một cách phù hợp: thay vì chỉ sử dụng tọa độ x của các điểm để xác định thứ tự, chúng ta sử dụng thứ tự chữ cái. Tức là, trước hết chúng ta sắp xếp theo tọa độ x, và nếu các đểm có cùng tọa độ x thì chúng ta xếp chúng theo tọa độ y. Một trường hợp đặc biệt mà chúng ta đã bỏ qua là xác định ba điểm tạo ra vòng trái hay vòng phải trên một đường thẳng. Trong trường hợp này, điểm giữa không nên xuất hiện trên bao lồi, vì vậy các điểm thẳng hàng phải được xem là tạo vòng trái. Nói cách khác, chúng ta nên sử dụng một phép kiểm tra trả về kết quả đúng nếu ba điểm tạo ra	

otherwise. (Note that this is simpler than the test required in the previous algorithm when there were collinear points.) With these modifications the algorithm correctly computes the convex hull: the first scan computes the upper hull, which is now defined as the part of the convex hull running from the lexicographically smallest vertex to the lexicographically largest vertex, and the second scan computes the remaining part of the convex hull. What does our algorithm do in the presence of rounding errors in the floating point arithmetic? When such errors occur, it can happen that a point is removed from the convex hull although it should be there, or that a point inside the real convex hull is not removed. But the structural integrity of the algorithm is unharmed: it will compute a closed polygonal chain. After all, the output is a list of points that we can interpret as the clockwise listing of the vertices of a polygon, and any three consecutive points form a right turn or, because of the rounding errors, they almost	một vòng phải, và sai trong trường hợp ngược lại. (Lưu ý rằng phép kiểm tra này đơn giản hơn phép kiểm tra cần thiết trong thuật toán trước đây khi có các điểm thẳng hàng.) Với những thay đổi này, thuật toán sẽ tính chính xác bao lồi: lần quét đầu tiên tính toán bao trên, hiện tại nó được định nghĩa là phần của bao lồi chạy từ đỉnh nhỏ nhất theo thứ tự chữ cái đến đỉnh lớn nhất theo thứ tự chữ cái, và lần quét thứ hai tính toán phần còn lại của bao lồi. Thuật toán của chúng ta làm gì khi có sự hiện diện của sai số làm tròn trong số học dấu chấm động? Khi những sai số như thế xảy ra, có khả năng một điểm nào đó bị loại bỏ khỏi bao lồi mặc dù sự hiện diện của nó ở đó là hợp lý, hoặc điểm bên trong bao lồi thực không được loại bỏ. Nhưng tính toàn vẹn cấu trúc của thuật toán không bị ảnh hưởng gì: nó sẽ tính toán một chuỗi đa giác khép kín. Sau cùng, đầu ra là một danh sách các điểm mà chúng ta có thể xem là một danh sách các đỉnh của đa giác theo chiều kim đồng hồ, và bất kỳ ba	

form a right turn. Moreover, no point in P can be far outside the computed hull. The only problem that can still occur is that, when three points lie very close together, a turn that is actually a sharp left turn can be interpretated as a right turn. This might result in a dent in the resulting polygon. A way out of this is to make sure that points in the input that are very close together are considered as being the same point, for example by rounding. Hence, although the result need not be exactly correct—but then, we cannot hope for an exact result if we use inexact arithmetic—it does make sense. For many applications this is good enough. Still, it is wise to be careful in the implementation of the basic test to avoid errors as much as possible. We conclude with the following theorem: Theorem 1.1 The convex hull of a set of n points in the plane can be computed in O(nlogn) time. Proof. We will prove the correctness of the	điểm liên tiếp nào tạo thành một vòng phải, hoặc do các sai số làm tròn, chúng gần như tạo thành một vòng phải. Hơn nữa, không có điểm nào thuộc P nằm xa bên ngoài bao được tính toán. Vấn đề duy nhất còn lại là, khi ba điểm nằm rất gần nhau, một vòng thực sự là vòng trái rõ nét có thể được xem là vòng phải. Điều này có thể dẫn đến một vết lõm trong đa giác cuối cùng. Chúng ta có thể giải quyết vấn đề này bằng cách đảm bảo rằng các điểm ở đầu vào rất gần nhau được xem là cùng một điểm, chẳng hạn như bằng cách làm tròn. Do đó, mặc dù kết quả không cần chính xác tuyệt đối -nhưng sau đó, chúng ta không thể mong đợi một kết quả chính xác nếu chúng ta dùng số học không chính xác-nó không có nghĩa. Đối với nhiều ứng dụng, điều này đã đáp ứng được. Tuy nhiên, cẩn thận trong quá trình thực hiện các phép kiểm tra cơ bản để tránh các sai số ở mức tối thểu là một việc làm khôn ngoan. Chúng ta kết luận với định lý sau đây: Định lý 1.1 Bao lồi của một tập hợp n điểm trong mặt phẳng có thể được tính trong thời gian O (nlogn).	

computation of the upper hull; the lower hull computation can be proved correct using similar arguments. The proof is by induction on the number of point treated. Before the for-loop starts, the list Lupper contains the points p1 and p2, which trivially form the upper hull of {p1, p2}. Now suppose that Lupper contains the upper hull vertices of {p1,..., pi-1} and consider the addition of pi. After the execution of the while-loop and because of the induction hypothesis, we know that the points in Lupper form a chain that only makes right turns. Moreover, the chain starts at the lexicographically smallest point of {p1,..., pi} and ends at the lexicographically largest point, namely pi. If we can show that all points of {p1,..., pi} that are not in Lupper are below the chain, then Lupper contains the correct points. By induction we know there is no point above the chain that we had before pi was added. Since the old chain lies below the new chain, the only possibility for a point to lie above the new chain is if it lies in the vertical slab between pi-1 and pi. But this is not possible, since such a	Chứng minh. Chúng ta sẽ chứng minh sự chính xác của quy trình tính bao trên; chúng ta cũng có thể đánh giá tính chính xác của quy trình tính bao dưới bằng lập luận tương tự. Chứng minh được thực hiện bằng phương pháp quy nạp dựa trên số điểm được xét. Trước khi vòng lặp for bắt đầu, danh sách Lupper chứa các điểm p1 và p2, thường tạo thành bao trên của {p1, p2}. Bây giờ giả sử Lupper chứa các đỉnh bao trên của {p1, ..., pi-1} và xét việc thêm vào pi. Sau khi thực hiện vòng lặp while và vì giả thuyết quy nạp, chúng ta biết rằng các điểm trong Lupper tạo thành một chuỗi chỉ tạo ra các vòng phải. Hơn nữa, chuỗi bắt đầu tại điểm nhỏ nhất theo thứ tự chữ cái của {p1, ..., pi} và kết thúc tại điểm lớn nhất theo thứ tự chữ cái, cụ thể là pi. Nếu chúng ta có thể chứng minh rằng tất cả các điểm {p1, ..., pi} không nằm trong Lupper nằm bên dưới chuỗi, thì Lupper chứa các điểm chính xác. Bằng phương pháp quy nạp, chúng ta biết được không có điểm nào ở trên chuỗi mà chúng ta có trước khi thêm pi vào. Bởi vì chuỗi cũ nằm bên dưới chuỗi mới, khả năng duy nhất để một điểm nằm bên trên chuỗi mới là nó phải nằm trong các thanh dọc giữa pi-1	

point would be in between pi-1 and pi in the lexicographical order. (You should verify that a similar argument holds if pi-1 and pi, or any other points, have the same x-coordinate.) To prove the time bound, we note that sorting the points lexicographically can be done in O(nlog n) time. Now consider the computation of the upper hull. The for-loop is executed a linear number of times. The question that remains is how often the while-loop inside it is executed. For each execution of the for-loop the while-loop is executed at least once. For any extra execution a point is deleted from the current hull. As each point can be deleted only once during the construction of the upper hull, the total number of extra executions over all for-loops is bounded by n. Similarly, the computation of the lower hull takes O(n) time. Due to the sorting step, the total time required for computing the convex hull is O(nlogn). EO The final convex hull algorithm is simple to describe and easy to implement. It only requires lexicographic sorting and a test whether three consecutive	và pi. Nhưng điều này không thể xảy ra, bởi vì một điểm như thế sẽ nằm giữa pi-1 và pi theo thứ tự bảng chữ cái. (Bạn nên xác nhận lập luận tương tự đúng nếu pi-1 và pi, hoặc bất kỳ điểm khác, có cùng tọa độ x.) Để chứng minh ràng buộc thời gian, chúng ta chú ý rằng việc sắp xếp các điểm theo thứ tự chữ cái có thể được thực hiện trong khoảng thời gian O(NLog n). Bây giờ xét quá trình tính toán bao trên. Vòng lặp for được thực hiện một số lần tuyến tính. Vấn đề còn lại là vòng lặp bên trong nó được thực hiện bao lâu một lần. Mỗi lần thực thi vòng lặp for, vòng lặp while được thực thi ít nhất một lần. Đối với những lần thực thi bổ sung, một điểm được phát hiện từ bao hiện tại. Bởi vì mỗi điểm chỉ có thể được phát hiện một lần trong quá trình xây dựng bao trên, tổng số lần thực thi phụ trong tất cả các vòng lặp for có cận là n. Tương tự, tính toán bao dưới mất khoảng thời gian O (n). Do các bước sắp xếp, tổng thời gian cần thiết để tính bao lồi là O (nlogn). EO Thuật toán bao lồi cuối cùng đơn giản và dễ thực hiện. Nó chỉ đòi hỏi sắp xếp theo thứ tự chữ cái và	

points make a right turn. From the original definition of the problem it was far from obvious that such an easy and efficient solution would exist. 1.2 Degeneracies and Robustness As we have seen in the previous section, the development of a geometric algorithm often goes through three phases. In the first phase, we try to ignore everything that will clutter our understanding of the geometric concepts we are dealing with. Sometimes collinear points are a nuisance, sometimes vertical line segments are. When first trying to design or understand an algorithm, it is often helpful to ignore these degenerate cases. In the second phase, we have to adjust the algorithm designed in the first phase to be correct in the presence of degenerate cases. Beginners tend to do this by adding a huge number of case distinctions to their algorithms. In many situations there is a better way. By considering the geometry of the problem again, one can often integrate special cases with the general case. For example, in the	kiểm tra xem ba điểm liên tiếp có tạo thành vòng phải hay không. Từ định nghĩa ban đầu của bài toán, chúng ta dễ dàng thấy được khả năng tồn tại của một cách giải hiệu quả và đơn giản như thế. 1.2 Sự suy biến và tính mạnh mẽ Như chúng ta đã thấy trong phần trước, việc xây dựng thuật toán hình học thường bao gồm ba giai đoạn. Trong giai đoạn đầu, chúng ta cố gắng bỏ qua mọi thứ có khả năng làm nhiễu những kiến thức hình học của chúng ta về vấn đề đang xét. Đôi khi, các điểm thẳng hàng lại gây phiền toái, các đoạn thẳng đứng cũng vậy. Khi lần đầu tiên thiết kế và tìm hiểu về một thuật toán, chúng ta nên bỏ qua những trường hợp suy biến này. Trong giai đoạn hai, chúng ta phải điều chỉnh các thuật toán được thiết kế trong giai đoạn đầu tiên cho chính xác hơn khi có các trường hợp suy biến. Những người mới bắt đầu có khuynh hướng phân chia bài toán thành nhiều trường hợp khi xây dựng thuật toán. Tuy nhiên, trong một số trường hợp, chúng ta lại có cách khác tốt hơn. Một lần nữa, qua việc xét tính chất hình học	

convex hull algorithm we only had to use the lexicographical order instead of the order on x-coordinate to deal with points with equal x-coordinate. For most algorithms in this book we have tried to take this integrated approach to deal with special cases. Still, it is easier not to think about such cases upon first reading. Only after understanding how the algorithm works in the general case should you think about degeneracies. If you study the computational geometry literature, you will find that many authors ignore special cases, often by formulating specific assumptions on the input. For example, in the convex hull problem we could have ignored special cases by simply stating that we assume that the input is such that no three points are collinear and no two points have the same x-coordinate. From a theoretical point of view, such assumptions are usually justified: the goal is then to establish the computational complexity of a problem and, although it is tedious to work out the	của bài toán, chúng ta thường có thể kết hợp các trường hợp đặc biệt với trường hợp tổng quát. Ví dụ, trong thuật toán bao lồi chúng ta chỉ cần dùng thứ tự bảng chữ cái thay cho thứ tự theo tọa độ x để xét các điểm có tọa độ x như nhau. Đối với đa số các thuật toán trong sách này, chúng ta cần thử phương pháp tiếp cận tích hợp này để xét các trường hợp đặc biệt.Tuy nhiên, trong lần đọc đầu tiên, chúng ta không thể nghĩ ra được các trường hợp như thế. Chỉ sau khi hiểu biết về cách thức hoạt động của thuật toán trong trường hợp tổng quát, chúng ta mới có thể nghĩ ra được các trường hợp suy biến. Nếu bạn nghiên cứu các tài liệu hình học tính toán, bạn sẽ thấy rằng nhiều tác giả bỏ qua các trường hợp đặc biệt, thông qua việc phát biểu các giả thuyết đặc biệt ở đầu vào. Ví dụ, trong bài toán bao lồi, chúng ta đã bỏ qua các trường hợp đặc biệt bằng cách phát biểu đơn giản rằng chúng ta giả sử rằng đầu vào không có ba điểm thẳng hàng và không có hai điểm nào có cùng tọa độ x. Từ quan điểm lý thuyết, giả thuyết này có vẻ chấp nhận được: mục tiêu là để điều khiển mức độ phức tạp của bài toán và, mặc dù xét từng trường hợp chi tiết	

details, degenerate cases can almost always be handled without increasing the asymptotic complexity of the algorithm. But special cases definitely increase the complexity of the implementations. Most researchers in computational geometry today are aware that their general position assumptions are not satisfied in practical applications and that an integrated treatment of the special cases is normally the best way to handle them. Furthermore, there are general techniques—so-called symbolic perturbation schemes—that allow one to ignore special cases during the design and implementation, and still have an algorithm that is correct in the presence of degeneracies.	như thế rất tẻ nhạt, các trường hợp suy biến luôn luôn có thể được xử lý mà không tăng tính phức tạp tiệm cận của thuật toán. Nhưng các trường hợp đặc biệt chắc chắn tăng sự phức tạp trong việc thực thi. Ngày nay, hầu hết các nhà nghiên cứu trong hình học tính toán đều nhận thức được rằng các giả thuyết tổng quát hóa của họ không phù hợp với các ứng dụng thực tế và cách tiếp cận tích hợp các trường hợp đặc biệt thường là cách giải quyết tốt nhất. Hơn nữa, có những kỹ thuật tổng quát, được gọi là sơ đồ nhiễu loạn ký tự, cho phép chúng ta bỏ qua các trường hợp đặc biệt trong thiết kế và thực thi, và vẫn còn có một thuật toán chính xác khi có suy biến.	
The third phase is the actual implementation. Now one needs to think about the primitive operations, like testing whether a point lies to the left, to the right, or on a directed line. If you are lucky you have a geometric software library available that contains the operations you need, otherwise you must implement them yourself.	Giai đoạn thứ ba là thực thi trong thực tế. Bây giờ, chúng phải nghĩ đến các phép toán cơ bản, chẳng hạn như kiểm tra xem một điểm nằm bên trái, bên phải, hay nằm trên một đường thẳng có hướng. Nếu bạn may mắn, bạn có sẵn một thư viện phần mềm hình học chứa các phép toán cần thiết, ngược lại, bạn phải tự thực hiện chúng.	
Another issue that arises in the implementation phase is		

that the assumption of doing exact arithmetic with real numbers breaks down, and it is necessary to understand the consequences. Robustness problems are often a cause of frustration when implementing geometric algorithms. Solving robustness problems is not easy. One solution is to use a package providing exact arithmetic (using integers, rationals, or even algebraic numbers, depending on the type of problem) but this will be slow. Alternatively, one can adapt the algorithm to detect inconsistencies and take appropriate actions to avoid crashing the program. In this case it is not guaranteed that the algorithm produces the correct output, and it is important to establish the exact properties that the output has. This is what we did in the previous section, when we developed the convex hull algorithm: the result might not be a convex polygon but we know that the structure of the output is correct and that the output polygon is very close to the convex hull. Finally, it is possible to predict, based on the input, the precision in the number representation required to solve the problem	Một vấn đề khác phát sinh trong giai đoạn thực thi là, giả thiết thực hiện số học chính xác với các số thực bị phá vỡ, và chúng ta cần biết về những hệ quả của nó. Các bài toán mạnh mẽ thường gây thất vọng khi thực thi các thuật toán hình học. Giải các bài toán mạnh mẽ không dễ. Một giải pháp cho vấn đề này là sử dụng gói phần mềm cung cấp số học chính xác (sử dụng số nguyên, số hữu tỷ, hoặc thậm chí số đại số, tùy thuộc vào loại bài toán) nhưng nó sẽ chậm. Ngoài ra, người ta có thể thay đổi các thuật toán cho phù hợp để phát hiện các mâu thuẫn và có những hành động thích hợp, tránh làm hỏng chương trình. Trong trường hợp này, chúng ta cũng chưa thể đảm bảo rằng thuật toán tạo ra đầu ra chính xác, và điều quan trọng là cần phải thiết lập đúng các tính chất mà đầu ra có. Đây là những công việc chúng ta đã làm trong phần trước, khi chúng ta xây dựng thuật toán bao lồi: kết quả có thể không phải là một đa giác lồi nhưng chúng ta biết rằng cấu trúc của đầu ra chính xác và đa giác đầu ra gần như là bao lồi. Cuối cùng, dựa trên đầu vào, nó có thể dự đoán độ chính xác trong biểu diễn số cần thiết để giải một bài toán chính xác.	

correctly. Which approach is best depends on the application. If speed is not an issue, exact arithmetic is preferred. In other cases it is not so important that the result of the algorithm is precise. For example, when displaying the convex hull of a set of points, it is most likely not noticeable when the polygon deviates slightly from the true convex hull. In this case we can use a careful implementation based on floating point arithmetic. In the rest of this book we focus on the design phase of geometric algorithms; we won't say much about the problems that arise in the implementation phase. 1.3 Application Domains As indicated before, we have chosen a motivating example application for every geometric concept, algorithm, or data structure introduced in this book. Most of the applications stem from the areas of computer graphics, robotics, geographic information systems, and CAD/CAM. For those not familiar with these fields, we	Cách nào là tốt nhất còn phụ thuộc vào ứng dụng. Nếu tốc độ không phải là một vấn đề, số học chính xác được ưu tiên hơn. Trong các trường hợp khác, người ta không quá quan trọng mức chính xác của thuật toán. Ví dụ, khi hiển thị bao lồi của một tập các điểm, việc đa giác hơi lệch so với bao lồi thực sự không quá quan trọng. Trong trường hợp này, chúng ta có thể sử dụng một phương pháp thực thi cẩn thận dựa trên số học dấu chấm động. Trong phần còn lại của cuốn sách này, chúng tôi tập trung vào giai đoạn thiết kế các thuật toán hình học, chúng ta sẽ không đề cập nhiều đến các vấn đề phát sinh ở giai đoạn thực thi. 1.3 Các lĩnh vực ứng dụng Như đã trình bày trước đây, chúng tôi đã chọn một ví dụ có tính chất động cơ cho mỗi khái niệm hình học, thuật toán hoặc cấu trúc dữ liệu được giới thiệu trong sách này. Hầu hết các ứng dụng xuất phát từ các lĩnh vực đồ họa máy tính, robot, hệ thống thông tin địa lý, và CAD / CAM. Đối với những người	

give a brief description of the areas and indicate some of the geometric problems that arise in them.	không quen thuộc với các lĩnh vực này, chúng tôi đưa ra mô tả ngắn gọn về các lĩnh vực và chỉ ra một số vấn đề hình học phát sinh trong những lĩnh vực đó.	
Computer graphics. computer graphics is concerned with creating images of modeled scenes for display on a computer screen, a printer, or other output device. The scenes vary from simple two-dimensional drawings—consisting of lines, polygons, and other primitive objects—to realistic-looking 3-dimensional scenes including light sources, textures, and so on. The latter type of scene can easily contain over a million polygons or curved surface patches.	Đồ họa máy tính. đồ họa máy tính có liên quan với việc tạo ra các hình ảnh của các phong cảnh được mô hình hóa để hiển thị trên màn hình máy tính, máy in, hoặc các thiết bị đầu ra khác. Các phong cảnh có thể là các hình vẽ hai chiều đơn giản như đường thẳng, đa giác, và các đối tượng cơ bản khác đến các phong cảnh 3 chiều trong như thực như các nguồn ánh sáng, kết cấu, và v.v…. Loại phong cảnh thứ hai có thể chứa trên một triệu đa giác và các phần bề mặt cong.	
Because scenes consist of geometric objects, geometric algorithms play an important role in computer graphics.	Bởi vì phong cảnh bao gồm các đối tượng hình học, các thuật toán hình học đóng vai trò quan trọng trong đồ họa máy tính.	
For 2-dimensional graphics, typical questions involve the intersection of certain primitives, determining the primitive pointed to with the mouse, or determining the subset of primitives that lie within a particular region. Chapters 6,10, and 16 describe techniques useful for some of these problems.	Đối với đồ họa 2 chiều, các vấn đề đặt ra có thể là giao của các đơn vị đồ họa nào đó, xác định các đơn vị đồ họa được trỏ tới với chuột, hoặc xác định tập hợp con của các đơn vị đồ họa nằm trong một khu vực cụ thể. Chương 6,10, và 16 mô tả các kỹ thuật hữu ích cho những vấn đề này.	
When dealing with 3-		

dimensional problems the geometric questions become more complex. A crucial step in displaying a 3-dimensional scene is hidden surface removal: determine the part of a scene visible from a particular viewpoint or, in other words, discard the parts that lie behind other objects. In Chapter 12 we study one approach to this problem. To create realistic-looking scenes we have to take light into account. This creates many new problems, such as the computation of shadows. Hence, realistic image synthesis requires complicated display techniques, like ray tracing and radiosity. When dealing with moving objects and in virtual reality applications, it is important to detect collisions between objects. All these situations involve geometric problems. Robotics. The field of robotics studies the design and use of robots. As robots are geometric objects that operate in a 3-dimensional space—the real world—it is obvious that geometric problems arise at many places. At the beginning of this chapter we already	Khi xét các bài toán ba chiều, các vấn đề hình học trở nên phức tạp hơn. Một bước quan trọng trong việc hiển thị một phong cảnh 3 chiều là loại bỏ bề mặt ẩn: xác định phần của phong cảnh có thể thấy được từ một góc nhìn cụ thể hay, nói cách khác, loại bỏ các phần nằm phía sau các đối tượng khác. Trong chương 12, chúng ta sẽ nghiên cứu một phương pháp để tiếp cận vấn đề này. Để tạo ra những phong cảnh giống với thực tế, chúng ta phải tính đến yếu tố ánh sáng. Điều này làm nảy sinh nhiều vấn đề mới, chẳng hạn như tính toán độ tối. Do đó, tổng hợp hình ảnh thực tế đòi hỏi các kỹ thuật hiển thị phức tạp, như dò tia sáng và tính toán sự va đập của ánh sáng. Khi xét các vật thể chuyển động và trong các ứng dụng thực tế ảo, việc phát hiện va chạm giữa hai vật thể rất quan trọng. Tất cả những tình huống này đều liên quan đến các bài toán hình học. Robot. Lĩnh vực robot học nghiên cứu việc thiết kế và sử dụng robot. Vì robot là các đối tượng hình học hoạt động trong không gian 3 chiều –thế giới thực-nên tất nhiên sẽ có nhiều bài toán hình học nảy sinh ở đây. Khi bắt đầu chương này,	

introduced the motion planning problem, where a robot has to find a path in an environment with obstacles.	chúng ta đã giới thiệu bài toán hoạch định chuyển động, trong đó robot phải tìm đường đi qua các vật cản.	

In Chapters 13 and 15 we study some simple cases of motion planning. Motion planning is one aspect of the more general problem of task planning. One would like to give a robot high-level tasks—"vacuum the room"—and let the robot figure out the best way to execute the task. This involves planning motions, planning the order in which to perform subtasks, and so on.

Other geometric problems occur in the design of robots and work cells in which the robot has to operate. Most industrial robots are robot arms with a fixed base. The parts operated on by the robot arm have to be supplied in such a way that the robot can easily grasp them. Some of the parts may have to be immobilized so that the robot can work on them. They may also have to be turned to a known orientation before the robot can work on them. These are all geometric problems, sometimes with a kinematic component. Some of the algorithms described in this book are applicable in such problems. For example, the smallest enclosing disc

problem, treated in Section 4.7, can be used for optimal placement of robot arms.

Geographic information systems. A geographic information system, or GIS for short, stores geographical data like the shape of countries, the height of mountains, the course of rivers, the type of vegetation at different locations, population density, or rainfall. They can also store human-made structures such as cities, roads, railways, electricity lines, or gas pipes. A GIS can be used to extract information about certain regions and, in particular, to obtain information about the relation between different types of data. For example, a biologist may want to relate the average rainfall to the existence of certain plants, and a civil engineer may need to query a GIS to determine whether there are any gas pipes underneath a lot where excavation works are to be performed.

As most geographic information concerns properties of points and regions on the earth's surface, geometric problems occur in abundance here. Moreover,

the amount of data is so large that efficient algorithms are a must. Below we mention the GIS-related problems treated in this book.

A first question is how to store geographic data. Suppose that we want to develop a car guidance system, which shows the driver at any moment where she is. This requires storing a huge map of roads and other data. At every moment we have to be able to determine the position of the car on the map and to quickly select a small portion of the map for display on the on-board computer. Efficient data structures are needed for these operations. Chapters 6, 10, and 16 describe computational geometry solutions to these problems.

The information about the height in some mountainous terrain is usually only available at certain sample points. For other positions we have to obtain the heights by interpolating between nearby sample points. But which sample points should we choose? Chapter 9 deals with this problem.

The combination of different types of data is one of the most important operations in

a GIS. For example, we may want to check which houses lie in a forest, locate all bridges by checking where roads cross rivers, or determine a good location for a new golf course by finding a slightly hilly, rather cheap area not too far from a particular town. A GIS usually stores different types of data in separate maps. To combine the data we have to overlay different maps. Chapter 2 deals with a problem arising when we want to compute the overlay. Finally, we mention the same example we gave at the beginning of this chapter: the location of the nearest public phone (or hospital, or any other facility). This requires the computation of a voronoi diagram, a structure studied in detail in Chapter 7. CAD/CAM. Computer aided design (CAD) concerns itself with the design of products with a computer. The products can vary from printed circuit boards, machine parts, or furniture, to complete buildings. In all cases the resulting product is a geometric entity and, hence,		

it is to be expected that all sorts of geometric problems appear. Indeed, CAD packages have to deal with intersec-tions and unions of objects, with decomposing objects and object boundaries into simpler shapes, and with visualizing the designed products. To decide whether a design meets the specifications certain tests are needed. Often one does not need to build a prototype for these tests, and a simulation suffices. Chapter 14 deals with a problem arising in the simulation of heat emission by a printed circuit board. Once an object has been designed and tested, it has to be manufactured. Computer aided manufacturing (CAM) packages can be of assistance here. CAM involves many geometric problems. Chapter 4 studies one of them. A recent trend is design for assembly, where assembly decisions are already taken into account during the design stage. A cad system supporting this would allow	[redacted]	

designers to test their design for feasibility, answering questions like: can the product be built easily using a certain manufacturing process? Many of these questions require geometric algorithms to be answered.

Other applications domains. There are many more application domains where geometric problems occur and geometric algorithms and data structures can be used to solve them.

For example, in molecular modeling, molecules are often represented by collections of intersecting balls in space, one ball for each atom. Typical questions are to compute the union of the atom balls to obtain the molecule surface, or to compute where two molecules can touch each other.

Another area is pattern recognition. Consider for example an optical character recognition system. Such a system scans a paper with text on it with the goal of recognizing the text characters. A basic step is to match the image of a character against a collection of stored characters to find the one that best fits it. This

leads to a geometric problem: given two geometric objects, determine how well they resemble each other.

Even certain areas that at first sight do not seem to be geometric can benefit from geometric algorithms, because it is often possible to formulate non- 12 geometric problem in geometric terms. In Chapter 5, for instance, we will see how records in a database can be interpreted as points in a higher-dimensional space, and we will present a geometric data structure such that certain queries on the records can be answered efficiently.

We hope that the above collection of geometric problems makes it clear that computational geometry plays a role in many different areas of computer science. The algorithms, data structures, and techniques described in this book will provide you with the tools needed to attack such geometric problems successfully.

1.4 Notes and Comments

Every chapter of this book ends with a section entitled

Notes and Comments.

These sections indicate where the results described in the chapter came from, indicate generalizations and improvements, and provide references. They can be skipped but do contain useful material for those who want to know more about the topic of the chapter. More information can also be found in the Handbook of Computational Geometry [331] and the Handbook of Discrete and Computational Geometry [191].

In this chapter the geometric problem treated in detail was the computation of the convex hull of a set of points in the plane. This is a classic topic in computational geometry and the amount of literature about it is huge. The algorithm described in this chapter is commonly known as Graham's scan, and is based on a modification by Andrew [17] of one of the earliest algorithms by Graham [192]. This is only one of the many O(nlogn) algorithms available for solving the problem. A divide-and-conquer approach was given by preparata and Hong [322]. Also an incremental method exists

that inserts the points one by one in O(logn) time per insertion [321]. Overmars and van Leeuwen generalized this to a method in which points could be both inserted and deleted in O(log2 n) time [305]. Other results on dynamic convex hulls were obtained by Hershberger and Suri [211], Chan [83], and Brodal and Jacob [73].

Even though an Q(nlogn) lower bound is known for the problem [393] many authors have tried to improve the result. This makes sense because in many applications the number of points that appear on the convex hull is relatively small, while the lower bound result assumes that (almost) all points show up on the convex hull. Hence, it is useful to look at algorithms whose running time depends on the complexity of the convex hull. Jarvis [221] introduced a wrapping technique, often referred to as Jarvis's march, that computes the convex hull in O(h • n) time where h is the complexity of the convex hull. The same worst-case performance is achieved by the algorithm of Overmars and van Leeuwen [303], based on earlier work by

Bykat [79], Eddy [156], and Green and Silverman [193]. This algorithm has the advantage that its expected running time is linear for many distributions of points. Finally, Kirkpatrick and Seidel [238] improved the result to O(nlogh), and recently Chan [82] discovered a much simpler algorithm to achieve the same result.

The convex hull can be defined in any dimension. Convex hulls in 3-dimensional space can still be computed in O(nlogn) time, as we will see in Chapter 11. For dimensions higher than 3, however, the complexity of the convex hull is no longer linear in the number of points. See the notes and comments of Chapter 11 for more details.

In the past years a number of general methods for handling special cases have been suggested. These symbolic perturbation schemes perturb the input in such a way that all degeneracies disappear. However, the perturbation is only done symbolically. This technique was introduced by Edelsbrunner and Mucke [164] and later refined by Yap [397] and Emiris and Canny [172, 171]. Symbolic

perturbation relieves the programmer of the burden of degeneracies, but it has some drawbacks: the use of a symbolic perturbation library slows down the algorithm, and sometimes one needs to recover the "real result" from the "perturbed result", which is not always easy. These drawbacks led Burnikel et al. [78] to claim that it is both simpler (in terms of programming effort) and more efficient (in terms of running time) to deal directly with degenerate inputs. Robustness in geometric algorithms is a topic that has recently received a lot of interest. Most geometric comparisons can be formulated as computing the sign of some determinant. A possible way to deal with the inexactness in floating point arithmetic when evaluating this sign is to choose a small threshold value e and to say that the determinant is zero when the outcome of the floating point computation is less than e. When implemented naively, this can lead to inconsistencies (for instance, for three points a, b, c we may decide that a = b and b = c but a = c) that cause		

the program to fail. Guibas et al. [198] showed that combining such an approach with interval arithmetic and backwards error analysis can give robust algorithms. Another option is to use exact arithmetic. Here one computes as many bits of the determinant as are needed to determine its sign. This will slow down the computation, but techniques have been developed to keep the performance penalty relatively small [182, 256, 395]. Besides these general approaches, there have been a number papers dealing with robust computation in specific problems [34, 37, 81, 145, 180, 181, 219, 279].

We gave a brief overview of the application domains from which we took our examples, which serve to show the motivation behind the various geometric notions and algorithms studied in this book. Below are some references to textbooks you can consult if you want to know more about the application domains. Of course there are many more good books about these domains than the few we mention.

There is a large number of books on computer graphics.

The book by Foley et al. [179] is very extensive and generally considered one of the best books on the topic. Other good books are the ones by Shirley et al. [359] and Watt [381].

An extensive overview of robotics and the motion planning problem can be found in the book of Choset et al. [127], and in the somewhat older books of Latombe [243] and Hopcroft, Schwartz, and Sharir [217]. More information on geometric aspects of robotics is provided by the book of Selig [348].

There is a large collection of books about geographic information systems, but most of them do not consider algorithmic issues in much detail. Some general textbooks are the ones by DeMers [140], Longley et al. [257], and Worboys and Duckham [392]. Data structures for spatial data are described extensively in the book of Samet [335].

The books by Faux and Pratt [175], Mortenson [285], and Hoffmann [216] are good introductory texts on CAD/CAM and geometric modeling.

1.5 Exercises 1.1 The convex hull of a set S is defined to be the intersection of all convex sets that contain S. For the convex hull of a set of points it was indicated that the convex hull is the convex set with smallest perimeter. We want to show that these are equivalent definitions. a. prove that the intersection of two convex sets is again convex. This implies that the intersection of a finite family of convex sets is convex as well. b. Prove that the smallest perimeter polygon P containing a set of points P is convex. c. Prove that any convex set containing the set of points P contains the smallest perimeter polygon P. 1.2 Let P be a set of points in the plane. Let P be the convex polygon whose vertices are points from P and that contains all points in P. prove that this polygon P is uniquely defined, and that it is the intersection of all convex sets containing P. 1.3 Let E be an unsorted set of n segments that are the edges of a convex polygon. Describe an O(nlog n) algorithm that computes from E a list containing all vertices	[redacted]	

of the polygon, sorted in clockwise order. 1.4 For the convex hull algorithm we have to be able to test whether a point r lies left or right of the directed line through two points p and q. Let p =(Px,Py), q =(qx, qy), and r =(rx, ry). a. Show that the sign of the determinant determines whether r lies left or right of the line. b. Show that |D| in fact is twice the surface of the triangle determined by p, q, and r. c. Why is this an attractive way to implement the basic test in algorithm CONVEXHULL? Give an argument for both integer and floating point coordinates. 1.5 Verify that the algorithm CONVEXHULL with the indicated modifications correctly computes the convex hull, also of degenerate sets of points. Consider for example such nasty cases as a set of points that all lie on one (vertical) line. 1.6 In many situations we need to compute convex hulls of objects other than points. a. Let S be a set of n line segments in the plane. Prove that the convex hull of S is		

exactly the same as the convex hull of the 2n endpoints of the segments. b. *Let P be a non-convex polygon. Describe an algorithm that computes the convex hull of P in O(n) time. Hint: Use a variant of algorithm ConvexHull where the vertices are not treated in lexicographical order, but in some other order. 1.7 Consider the following alternative approach to computing the convex hull of a set of points in the plane: We start with the rightmost point. This is the first point p1 of the convex hull. Now imagine that we start with a vertical line and rotate it clockwise until it hits another point p2. This is the second point on the convex hull. We continue rotating the line but this time around p2 until we hit a point p3. In this way we continue until we reach p1 again. a. Give pseudocode for this algorithm. b. What degenerate cases can occur and how can we deal with them? c. Prove that the algorithm correctly computes the convex hull. d. Prove that the algorithm can be implemented to run in time		

O(n • h), where h is the complexity of the convex hull. e. What problems might occur when we deal with inexact floating point arithmetic? 1.8 The O(nlog n) algorithm to compute the convex hull of a set of n points in the plane that was described in this chapter is based on the paradigm of incremental construction: add the points one by one, and update the convex hull after each addition. In this exercise we shall develop an algorithm based on another paradigm, namely divide-and-conquer. a. Let P1 and P2 be two disjoint convex polygons with n vertices in total. Give an O(n) time algorithm that computes the convex hull of P1 u P2. b. Use the algorithm from part a to develop an O(nlog n) time divide-and- conquer algorithm to compute the convex hull of a set of n points in the plane. 1.9 Suppose that we have a subroutine ConvexHull available for comput-ing the convex hull of a set of points in the plane. Its output is a list of convex hull vertices, sorted in clockwise order. Now let S = {x1, x2,...,xn} be a set of n		

numbers. Show that S can be sorted in O(n) time plus the time needed for one call to ConvexHull. Since the sorting problem has an Q.(nlogn) lower bound, this implies that the convex hull problem has an fì(nlogn) lower bound as well. Hence, the algorithm presented in Section 1.5 this chapter is asymptotically optimal. EXERCISES 1.10 Let S be a set of n (possibly intersecting) unit circles in the plane. We want to compute the convex hull of S. a. Show that the boundary of the convex hull of S consists of straight line segments and pieces of circles in S. b. Show that each circle can occur at most once on the boundary of the convex hull. c. Let S' be the set of points that are the centers of the circles in S. Show that a circle in S appears on the boundary of the convex hull if and only if the center of the circle lies on the convex hull of S'. d. Give an O(nlog n) algorithm for computing the convex hull of S. e. *Give an O(nlogn) algorithm for the case in which the circles in S have		

different radii. 2 Line Segment Intersection Thematic Map Overlay When you are visiting a country, maps are an invaluable source of information. They tell you where tourist attractions are located, they indicate the roads and railway lines to get	[redacted]	

there, they show small lakes, and so on. Unfortunately, they can also be a source of frustration, as it is often difficult to find the right information: even when you know the approximate position of a small town, it can still be difficult to spot it on the map. To make maps more readable, geographic information systems split them into several layers. Each layer is a thematic map, that is, it stores only one type of information. Thus there will be a layer storing the roads, a layer storing the cities, a layer storing the rivers, and so on. The theme of a layer can also be more abstract. For instance, there could be a layer for the population density, for average precipitation, habitat of the grizzly bear, or for vegetation. The type of geometric information stored in a layer can be very different: the layer for a road map could store the roads as collections of line segments (or curves, perhaps), the layer for cities could contain points labeled with city names, and the layer for vegetation could store a subdivision of the map into regions labeled with the type of vegetation.

Users of a geographic

information system can select one of the thematic maps for display. To find a small town you would select the layer storing cities, and you would not be distracted by information such as the names of rivers and lakes. After you have spotted the town, you probably want to know how to get there. To this end geographic information systems allow users to view an overlay of several maps—see Figure 2.1. Using an overlay of the road map and the map storing cities you can now figure out how to get to the town. When two or more thematic map layers are shown together, intersections in the overlay are positions of special interest. For example, when viewing the overlay of the layer for the roads and the layer for the rivers, it would be useful if the intersections were clearly marked. In this example the two maps are basically networks, and the intersections are points. In other cases one is interested in the intersection of complete regions. For instance, geographers studying the climate could be interested in finding regions where there is pine forest and the annual precipitation is

between 1000 mm and 1500 mm. These regions are the intersections of the regions labeled “pine forest” in the vegetation map and the regions labeled “1000-1500” in the precipitation map.

2.1 Line Segment Intersection

We first study the simplest form of the map overlay problem, where the two map layers are networks represented as collections of line segments. For example, a map layer storing roads, railroads, or rivers at a small scale. Note that curves can be approximated by a number of small segments. At first we won’t be interested in the regions induced by these line segments. Later we shall look at the more complex situation where the maps are not just networks, but subdivisions of the plane into regions that have an explicit meaning. To solve the network overlay problem we first have to state it in a geometric setting.

For the overlay of two networks the geometric situation is the following: given two sets of line segments, compute all intersections between a

segment from one set and a segment from the other. This problem specification is not quite precise enough yet, as we didn't define when two segments intersect. In particular, do two segments intersect when an endpoint of one of them lies on the other? In other words, we have to specify whether the input segments are open or closed. To make this decision we should go back to the application, the network overlay problem. Roads in a road map and rivers in a river map are represented by chains of segments, so a crossing of a road and a river corresponds to the interior of one chain intersecting the interior of another chain. This does not mean that there is an intersection between the interior of two segments: the intersection point could happen to coincide with an endpoint of a segment of a chain. In fact, this situation is not uncommon because windy rivers are represented by many small segments and coordinates of endpoints may have been rounded when maps are digitized. We conclude that we should define the segments to be closed, so that an endpoint of one segment lying on another

segment counts as an intersection.

To simplify the description somewhat we shall put the segments from the two sets into one set, and compute all intersections among the segments in that set. This way we certainly find all the intersections we want. We may also find intersections between segments from the same set. Actually, we certainly will, because in our application the segments from one set form a number of chains, and we count coinciding endpoints as intersections. These other intersections can be filtered out afterwards by simply checking for each reported intersection whether the two segments involved belong to the same set. So our problem specification is as follows: given a set S of n closed segments in the plane, report all intersection points among the segments in S.

This doesn't seem like a challenging problem: we can simply take each pair of segments, compute whether they intersect, and, if so, report their intersection point.

This brute-force algorithm clearly requires O(n2) time. In a sense this is optimal: when each pair of segments intersects any algorithm must take Q(n2) time, because it has to report all intersections. A similar example can be given when the overlay of two networks is considered. In practical situations, however, most segments intersect no or only a few other segments, so the total number of intersection points is much smaller than quadratic. It would be nice to have an algorithm that is faster in such situations. In other words, we want an algorithm whose running time depends not only on the number of segments in the input, but also on the number of intersection points. Such an algorithm is called an output-sensitive algorithm: the running time of the algorithm is sensitive to the size of the output. We could also call such an algorithm intersection-sensitive, since the number of intersections is what determines the size of the output. How can we avoid testing all pairs of segments for	[redacted]	

intersection? Here we must make use of the geometry of the situation: segments that are close together are candidates for intersection, unlike segments that are far apart. Below we shall see how we can use this observation to obtain an output-sensitive algorithm for the line segment intersection problem.

Let S := {s1, s2,...,sn} be the set of segments for which we want to compute all intersections. We want to avoid testing pairs of segments that are far apart. But how can we do this? Let's first try to rule out an easy case. Define the y-interval of a segment to be its orthogonal projection onto the y-axis. When the y-intervals of a pair of segments do not overlap—we could say that they are far apart in the y-direction—then they cannot intersect. Hence, we only need to test pairs of segments whose y-intervals overlap, that is, pairs for which there exists a horizontal line that intersects both segments. To find these pairs we imagine sweeping a line I downwards over the plane, starting from a position above all segments. While we sweep the

imaginary line, we keep track of all segments intersecting it—the details of this will be explained later—so that we can find the pairs we need. This type of algorithm is called a plane sweep algorithm and the line I is called the sweep line. The status of the sweep line is the set of segments intersecting it. The status changes while the sweep line moves downwards, but not continuously. Only at particular points is an update of the status required. We call these points the event points of the plane sweep algorithm. In this algorithm the event points are the endpoints of the segments. The moments at which the sweep line reaches an event point are the only moments when the algorithm actually does something: it updates the status of the sweep line and performs some intersection tests. In particular, if the event point is the upper endpoint of a segment, then a new segment starts intersecting the sweep		

line and must be added to the status. This segment is tested for intersection against the ones already intersecting the sweep line. If the event point is a lower endpoint, a segment stops intersecting the sweep line and must be deleted from the status. This way we only test pairs of segments for which there is a horizontal line that intersects both segments. Unfortunately, this is not enough: there are still situations where we test a quadratic number of pairs, whereas there is only a small number of intersection points. A simple example is a set of vertical segments that all intersect the X-axis. So the algorithm is not output-sensitive. The problem is that two segments that intersect the sweep line can still be far apart in the horizontal direction.

Let's order the segments from left to right as they intersect the sweep line, to include the idea of being close in the horizontal direction. We shall only test segments when they are adjacent in the horizontal ordering. This means that we only test any new segment against two segments, namely, the ones immediately left and right of the upper

endpoint. Later, when the sweep line has moved downwards to another position, a segment can become adjacent to other segments against which it will be tested. Our new strategy should be reflected in the status of our algorithm: the status now corresponds to the ordered sequence of segments intersecting the sweep line. The new status not only changes at endpoints of segments; it also changes at intersection points, where the order of the intersected segments changes. When this happens we must test the two segments that change position against their new neighbors. This is a new type of event point.

Before trying to turn these ideas into an efficient algorithm, we should convince ourselves that the approach is correct. We have reduced the number of pairs to be tested, but do we still find all intersections? In other words, if two segments si and sj intersect, is there always a position of the sweep line I where si and sj are adjacent along I?

Let's first ignore some nasty cases: assume that no segment is horizontal, that

any two segments intersect in at most one point—they do not overlap—, and that no three segments meet in a common point. Later we shall see that these cases are easy to handle, but for now it is convenient to forget about them. The intersections where an endpoint of a segment lies on another segment can easily be detected when the sweep line reaches the endpoint. So the only question is whether intersections between the interiors of segments are always detected.

Lemma 2.1 Let si and Sj be two non-horizontal segments whose interiors intersect in a single point p, and assume there is no third segment passing through p. Then there is an event point above p where si and Sj become adjacent and are tested for intersection.

Proof. Let I be a horizontal line slightly above p. If I is close enough to p then si and Sj must be adjacent along I. (To be precise, we should take I such that there is no event point on Í, nor in between Í and the horizontal line through p.) In other words, there is a position of the sweep line where si and

Sj are adjacent. On the other hand, si and Sj are not yet adjacent when the algorithm starts, because the sweep line starts above all line segments and the status is empty. Hence, there must be an event point q where Si and Sj become adjacent and are tested for intersection. □

So our approach is correct, at least when we forget about the nasty cases mentioned earlier. Now we can proceed with the development of the plane sweep algorithm. Let's briefly recap the overall approach. We imagine moving a horizontal sweep line I downwards over the plane. The sweep line halts at certain event points; in our case these are the endpoints of the segments, which we know beforehand, and the intersection points, which are computed on the fly. While the sweep line moves we maintain the ordered sequence of segments intersected by it. When the sweep line halts at an event point the sequence of segments changes and, depending on the type of event point, we have to take several actions to update the status and detect intersections.

When the event point is the

upper endpoint of a segment, there is a new segment intersecting the sweep line. This segment must be tested for intersection against its two neighbors along the sweep line. Only intersection points below the sweep line are important; the ones above the sweep line have been detected already. For example, if segments Si and Sk are adjacent on the sweep line, and a new upper endpoint of a segment Sj appears in between, then we have to test Sj for intersection with Si and Sk. If we find an intersection below the sweep line, we have found a new event point. After the upper endpoint is handled we continue to the next event point.

When the event point is an intersection, the two segments that intersect change their order. Each of them gets (at most) one new neighbor against which it is tested for intersection. Again, only intersections below the sweep line are still interesting. Suppose that four segments Sj, Sk, Sl, and Sm appear in this order on the sweep line when the intersection point of Sk and Sl is reached. Then Sk and Sl switch position and we must

test Sl and Sj for intersection below the sweep line, and also Sk and Sm. The new intersections that we find are, of course, also event points for the algorithm. Note, however, that it is possible that these events have already been detected earlier, namely if a pair becoming adjacent has been adjacent before.

When the event point is the lower endpoint of a segment, its two neighbors now become adjacent and must be tested for intersection. If they intersect below the sweep line, then their intersection point is an event point. (Again, this event could have been detected already.) Assume three segments sk, sl, and sm appear in this order on the sweep line when the lower endpoint of sl is encountered. Then sk and sm will become adjacent and we test them for intersection.

After we have swept the whole plane—more precisely, after we have treated the last event point—we have computed all intersection points. This is guaranteed by the following invariant, which holds at any time during the plane sweep: all intersection points above the sweep line have been

computed correctly.

After this sketch of the algorithm, it's time to go into more detail. It's also time to look at the degenerate cases that can arise, like three or more segments meeting in a point. We should first specify what we expect from the algorithm in these cases.

We could require the algorithm to simply report each intersection point once, but it seems more useful if it reports for each intersection point a list of segments that pass through it or have it as an endpoint. There is another special case for which we should define the required output more carefully, namely that of two partially overlapping segments, but for simplicity we shall ignore this case in the rest of this section. We start by describing the data structures the algorithm uses.

First of all we need a data structure—called the event queue—that stores the events. We denote the event queue by Q.

We need an operation that removes the next event that will occur from Q, and

returns it so that it can be treated. This event is the highest event below the sweep line. If two event points have the same y-coordinate, then the one with smaller X-coordinate will be returned. In other words, event points on the same horizontal line are treated from left to right. This implies that we should consider the left endpoint of a horizontal segment to be its upper endpoint, and its right endpoint to be its lower endpoint. You can also think about our convention as follows: instead of having a horizontal sweep line, imagine it is sloping just a tiny bit upward. As a result the sweep line reaches the left endpoint of a horizontal segment just before reaching the right endpoint. The event queue must allow insertions, because new events will be computed on the fly. Notice that two event points can coincide. For example, the upper endpoints of two distinct segments may coincide. It is convenient to treat this as one event point. Hence, an insertion must be able to check whether an event is already present in Q.

We implement the event

queue as follows. Define an order -< on the event points that represents the order in which they will be handled. Hence, if p and q are two event points then we have p -< q if and only if py > qy holds or py = qy and pX < qX holds. We store the event points in a balanced binary search tree, ordered according to -< With each event point p in Q we will store the segments starting at p, that is, the segments whose upper endpoint is p. This information will be needed to handle the event. Both operations—fetching the next event 24 and inserting an event—take O(log m) time, where m is the number of events
in Q. (We do not use a heap to implement the event queue, because we have to be able to test whether a given event is already present in Q.)

Second, we need to maintain the status of the algorithm. This is the ordered sequence of segments intersecting the sweep line. The status structure, denoted by T, is used to access the neighbors of a given segment S, so that they can be tested for intersection with S. The status structure must be dynamic: as

segments start or stop to intersect the sweep line, they must be inserted into or deleted from the structure. Because there is a well-defined order on the segments in the status structure we can use a balanced binary search tree as status structure. When you are only used to binary search trees that store numbers, this may be surprising. But binary search trees can store any set of elements, as long as there is an order on the elements.

In more detail, we store the segments intersecting the sweep line ordered in the leaves of a balanced binary search tree T. The left-to-right order of the segments along the sweep line corresponds to the left-to-right order of the leaves in T. We must also store information in the internal nodes to guide the search down the tree to the leaves. At each internal node, we store the segment from the rightmost leaf in its left subtree. (Alternatively, we could store the segments only in interior nodes. This will save some storage. However,

it is conceptually simpler to think about the segments in interior nodes as values to guide the search, not as data items. Storing the segments in the leaves also makes some algorithms simpler to describe.) Suppose we search in T for the segment immediately to the left of some point p that lies on the sweep line. At each internal node V we test whether p lies left or right of the segment stored at V. Depending on the outcome we descend to the left or right subtree of V, eventually ending up in a leaf. Either this leaf, or the leaf immediately to the left of it, stores the segment we are searching for. In a similar way we can find the segment immediately to the right of p, or the segments containing p. It follows that each update and neighbor search operation takes O(log n) time.

The event queue Q and the status structure T are the only two data structures we need. The global algorithm can now be described as follows.

Algorithm
FINDINTERSECTIONS(S)

Input. A set S of line segments in the plane.

Output. The set of intersection points among the segments in S, with for each intersection point the segments that contain it.

1. Initialize an empty event queue Q. Next, insert the segment endpoints into Q; when an upper endpoint is inserted, the corresponding segment should be stored with it.
2. Initialize an empty status structure T.
3. while Q is not empty
4. do Determine the next event point p in Q and delete it.
5. HANDLEEVENTPOINT(P)

We have already seen how events are handled: at endpoints of segments we have to insert or delete segments from the status structure T, and at intersection points we have to change the order of two segments. In both cases we also have to do intersection tests between segments that become neighbors after the

event. In degenerate cases—where several segments are involved in one event point—the details are a little bit more tricky. The next procedure describes how to handle event points correctly; it is illustrated in Figure 2.2.

Figure 2.2

An event point and the changes in the status structure
HANDLEEVENTPOINT(p)

1. Let U(p) be the set of segments whose upper endpoint is p; these segments are stored with the event point p. (For horizontal segments, the upper endpoint is by definition the left endpoint.)
2. Find all segments stored in T that contain p; they are adjacent in T. Let L(p) denote the subset of segments found whose lower endpoint is p, and let C(p) denote the subset of segments found that contain p in their interior.
3. if L(p) u U(p) u C(p) contains more than one segment
4. then Report p as an intersection, together with L(p), U(p), and C(p).

5. Delete the segments in L(p) u C(p) from T. 6. Insert the segments in U(p) u C(p) into T. The order of the segments in T should correspond to the order in which they are intersected by a sweep line just below p. If there is a horizontal segment, it comes last among all segments containing p. 7. Deleting and re-inserting the segments of C(p) reverses their order. *) 8. if U(p) u C(p) = 0 9. then Let sl and sr be the left and right neighbors of p in T. 10. findNewEvent(s , sr, p) 11. else Let s' be the leftmost segment of U(p) u C(p) in T. 12. Let sl be the left neighbor of s' in T. 13. FindNewEvent(s1 , s', p) 14. Let s" be the rightmost segment of U(p) u C(p) in T. 15. Let sr be the right neighbor of s" in T.		

16. FindNewEvent(s", sr, p)

Note that in lines 8-16 we assume that sl and sr actually exist. If they do not exist the corresponding steps should obviously not be performed.

The procedures for finding the new intersections are easy: they simply test two segments for intersection. The only thing we need to be careful about is, when we find an intersection, whether this intersection has already been handled earlier or not. When there are no horizontal segments, then the intersection has not been handled yet when the intersection point lies below the sweep line. But how should we deal with horizontal segments? Recall our convention that events with the same y-coordinate are treated from left to right. This implies that we are still interested in intersection points lying to the right of the current event point. Hence, the procedure FindNewEvent is defined as follows.

FindNewEvent(s1 , sr, p)

1. if sl and sr intersect below the sweep line, or on it and to the right of the current event point p, and the intersection is not yet present

as an event in Q

2. then Insert the intersection point as an event into Q.

What about the correctness of our algorithm? It is clear that FINDINTERSEC- TIONS only reports true intersection points, but does it find all of them? The next lemma states that this is indeed the case.

Lemma 2.2 Algorithm FindIntersections computes all intersection points and the segments that contain it correctly.

Proof. Recall that the priority of an event is given by its y-coordinate, and that when two events have the same y-coordinate the one with smaller x-coordinate is given higher priority. We shall prove the lemma by induction on the priority of the event points.

Let p be an intersection point and assume that all intersection points q with a higher priority have been computed correctly. We shall prove that p and the segments that contain p are computed correctly. Let U (p) be the set of segments that have p as their upper endpoint (or, for

horizontal segments, their left endpoint), let L(p) be the set of segments having p as their lower endpoint (or, for horizontal segments, their right endpoint), and let C(p) be the set of segments having p in their interior.	[redacted]	
First, assume that p is an endpoint of one or more of the segments. In that case p is stored in the event queue Q at the start of the algorithm. The segments from U (p) are stored with p, so they will be found. The segments from L(p) and C(p) are stored in T when p is handled, so they will be found in line 2 of HandleEventPoint. Hence, p and all the segments involved are determined correctly when p is an endpoint of one or more of the segments.	[redacted]	
Now assume that p is not an endpoint of a segment. All we need to show is that p will be inserted into Q at some moment. Note that all segments that are involved have p in their interior. Order these segments by angle around p, and let si and Sj be two neighboring segments. Following the proof of Lemma 2.1 we see that there	[redacted]	

is an event point with a higher priority than p such that Si and Sj become adjacent when q is passed.

In Lemma 2.1 we assumed for simplicity that si and Sj are non-horizontal, but it is straightforward to adapt the proof for horizontal segments. By induction, the event point q was handled correctly, which means that p is detected and stored into Q.

So we have a correct algorithm. But did we succeed in developing an output- sensitive algorithm? The answer is yes: the running time of the algorithm is O((n + k) log n), where k is the size of the output. The following lemma states an even stronger result: the running time is O((n +1)logn), where I is the number of intersections. This is stronger, because for one intersection point the output can consist of a large number of segments, namely in the case where many segments intersect in a common point.

Lemma 2.3 The running time of Algorithm FindIntersections for a set S of n line segments in the plane is O(n log n +1 log n),

where I is the number of intersection points of segments in S.

Proof. The algorithm starts by constructing the event queue on the segment endpoints. Because we implemented the event queue as a balanced binary search tree, this takes O(nlogn) time. Initializing the status structure takes constant time. Then the plane sweep starts and all the events are handled. To handle an event we perform three operations on the event queue Q: the event itself is deleted from Q in line 4 of FindIntersections, and there can be one or two calls to FindNewEvent, which may cause at most two new events to be inserted into Q. Deletions and insertions on Q take O(logn) time each. We also perform operations—insertions, deletions, and neighbor finding—on the status structure T, which take O(log n) time each. The number of operations is linear in the number m(p) := card(L(p) u U(p) u C(p)) of segments that are involved in the event. If we denote the sum of all m(p), over all event points p, by m, the running time of the algorithm is O(m log n).

It is clear that m = O(n + k), where k is the size of the output; after all, whenever m(p) > 1 we report all segments involved in the event, and the only events involving one segment are the endpoints of segments. But we want to prove that m = O(n +1), where I is the number of intersection points. To show this, we will interpret the set of segments as a planar graph embedded in the plane. (If you are not familiar with planar graph terminology, you should read the first paragraphs of Section 2.2 first.) Its vertices are the endpoints of segments and intersection points of segments, and its edges are the pieces of the segments connecting vertices. Consider an event point p. It is a vertex of the graph, and m(p) is bounded by the degree of the vertex. Consequently, m is bounded by the sum of the degrees of all vertices of our graph. Every edge of the graph contributes one to the degree of exactly two vertices (its endpoints), so m is bounded by 2ne, where ne is the number of edges of the graph. Let's bound ne in terms of n and I. By

definition, nv, the number of vertices, is at most 2n +1. It is well known that in planar graphs ne = O(nv), which proves our claim. But, for completeness, let us give the argument here. Every face of the planar graph is bounded by at least three edges—provided that there are at least three segments—and an edge can bound at most two different faces. Therefore nf, the number of faces, is at most 2ne/3. We now use Euler's formula, which states that for any planar graph with nv vertices, ne edges, and nf faces, the following relation holds:

nv — ne + nf > 2.

Equality holds if and only if the graph is connected. Plugging the bounds on nv and nf into this formula, we get

…….

So ne < 6n + 3I — 6, and m < 12n + 6I — 12, and the bound on the running time follows.

We still have to analyze the other complexity aspect, the amount of storage used by the algorithm. The tree T stores a segment at most once, so it uses O(n) storage. The size of Q can be larger, however.

The algorithm inserts intersection points in Q when they are detected and it removes them when they are handled. when it takes a long time before intersections are handled, it could happen that Q gets very large. Of course its size is always bounded by O(n+1), but it would be better if the working storage were always linear.	[redacted]	

There is a relatively simple way to achieve this: only store intersection points of pairs of segments that are currently adjacent on the sweep line. The algorithm given above also stores intersection points of segments that have been horizontally adjacent, but aren't anymore. By storing only intersections among adjacent segments, the number of event points in Q is never more than linear. The modification required in the algorithm is that the intersection point of two segments must be deleted when they stop being adjacent. These segments must become adjacent again before the intersection point is reached, so the intersection point will still be reported correctly. The total time taken by the algorithm remains O(n log n +1 log n). We obtain the following theorem:

Theorem 2.4 Let S be a set of n line segments in the plane. All intersection points in S, with for each intersection point the segments involved in it, can be reportedin O(nlogn + Ilogn) time and O(n) space, where I is the number of intersection points.

2.2 The Doubly-Connected

Edge List

we have solved the easiest case of the map overlay problem, where the two maps are networks represented as collections of line segments. In general, maps have a more complicated structure: they are subdivisions of the plane into labeled regions. A thematic map of forests in Canada, for instance, would be a subdivision of Canada into regions with labels such as “pine”, “deciduous”, “birch”, and “mixed”.

Before we can give an algorithm for computing the overlay of two subdivisions, we must develop a suitable representation for a subdivision. Storing a subdivision as a collection of line segments is not such a good idea. Operations like reporting the boundary of a region would be rather complicated. It is better

Figure 2.3 Types of forest in Canada

to incorporate structural, topological information: which segments bound a given region, which regions are adjacent, and so on.

The maps we consider are

planar subdivisions induced by planar embeddings of graphs. Such a subdivision is connected if the underlying graph is connected. The embedding of a node of the graph is called a vertex, and the embedding of an arc is called an edge. We only consider embeddings where every edge is a straight line segment. In principle, edges in a subdivision need not be straight. A subdivision need not even be a planar embedding of a graph, as it may have unbounded edges. In this section, however, we don't consider such more general subdivisions. We consider an edge to be open, that is, its endpoints—which are vertices of the subdivision—are not part of it. A face of the subdivision is a maximal connected subset of the plane that doesn't contain a point on an edge or a vertex. Thus a face is an open polygonal region whose boundary is formed by edges and vertices from the subdivision. The complexity of a subdivision is defined as the sum of the number of vertices, the number of edges, and the number of faces it consists of. If a vertex is the endpoint of an edge, then we say that the vertex and the

edge are incident. Similarly, a face and an edge on its boundary are incident, and a face and a vertex of its boundary are incident.

What should we require from a representation of a subdivision? An operation one could ask for is to determine the face containing a given point. This is definitely useful in some applications—indeed, in a later chapter we shall design a data structure for this—but it is a bit too much to ask from a basic representation. The things we can ask for should be more local. For example, it is reasonable to require that we can walk around the boundary of a given face, or that we can access one face from an adjacent one if we are given a common edge. Another operation that could be useful is to visit all the edges around a given vertex. The representation that we shall discuss supports these operations. It is called the doubly-connected edge list.

A doubly-connected edge list contains a record for each face, edge, and vertex of the

subdivision. Besides the geometric and topological information—to be described shortly—each record may also store additional information. For instance, if the subdivision represents a thematic map for vegetation, the doubly- connected edge list would store in each face record the type of vegetation of the corresponding region. The additional information is also called attribute information. The geometric and topological information stored in the doubly-connected edge list should enable us to perform the basic operations mentioned earlier. To be able to walk around a face in counterclockwise order we store a pointer from each edge to the next. It can also come in handy to walk around a face the other way, so we also store a pointer to the previous edge. An edge usually bounds two faces, so we need two pairs of pointers for it. It is convenient to view the different sides of an edge as two distinct half-edges, so that we have a unique next half-edge and previous half-edge for every half-edge. This also means that a half-edge bounds only one face. The two half-edges we get for a given edge are called twins.

Defining the next half-edge of a given half-edge with respect to a counterclockwise traversal of a face induces an orientation on each half-edge: it is oriented such that the face that it bounds lies to its left for an observer walking along the edge. Because half-edges are oriented we can speak of the origin and the destination of a half-edge. If a half-edge e has v as its origin and w as its destination, then its twin Twin(e) has w as its origin and v as its destination. To reach the boundary of a face we just need to store one pointer in the face record to an arbitrary half-edge bounding the face. Starting from that half-edge, we can step from each half-edge to the next and walk around the face.

what we just said does not quite hold for the boundaries of holes in a face: if they are traversed in counterclockwise order then the face lies to the right. It will be convenient to orient half-edges such that their face always lies to the same side, so we change the direction of traversal for the boundary of a hole to clockwise. Now a face always lies to the left of any half-

edge on its boundary.

Another consequence is that twin half-edges always have opposite orientations. The presence of holes in a face also means that one pointer from the face to an arbitrary half-edge on its boundary is not enough to visit the whole boundary: we need a pointer to a half-edge in every boundary component. If a face has isolated vertices that don't have any incident edge, we can store pointers to them as well. For simplicity we'll ignore this case.

Let's summarize. The doubly-connected edge list consists of three collections of records: one for the vertices, one for the faces, and one for the half-edges. These records store the following geometric and topological information:

- The vertex record of a vertex v stores the coordinates of v in a field called Coordinates(v). It also stores a pointer IncidentEdge(v) to an arbitrary half-edge that has v as its origin.
- The face record of a face f stores a pointer OuterComponent(f) to some half-edge on its outer

boundary. For the unbounded face this pointer is nil. It also stores a list InnerComponents(f), which contains for each hole in the face a pointer to some half-edge on the boundary of the hole.

■ The half-edge record of a half-edge e stores a pointer Origin(e) to its origin, a pointer Twin(e) to its twin half-edge, and a pointer IncidentFace(e) to the face that it bounds. We don't need to store the destination of an edge, because it is equal to Origin(Twin(e)). The origin is chosen such that IncidentFace(e) lies to the left of e when it is traversed from origin to destination. The half-edge record also stores pointers Next(e) and Prev(e) to the next and previous edge on the boundary of IncidentFace(e). Thus Next(e) is the unique half-edge on the boundary of IncidentFace(e) that has the destination of e as its origin, and Prev(e) is the unique half-edge on the boundary of IncidentFace(e) that has Origin(e) as its destination.

A constant amount of information is used for each vertex and edge. A face may require more storage, since the list InnerComponents(f)

has as many elements as there are holes in the face.
Because any half-edge is pointed to at most once from all InnerComponents(f) lists together, we conclude that the amount of storage is linear in the complexity of the subdivision.

An example of a doubly-connected edge list for a simple subdivision is given below. The two half-edges corresponding to an edge ei are labeled Êự and ei 2.

The information stored in the doubly-connected edge list is enough to perform the basic operations. For example, we can walk around the outer boundary of a given face f by following Next(e) pointers, starting from the half-edge OuterComponent(f). We can also visit all edges incident to a vertex v. It is a good exercise to figure out for yourself how to do this.

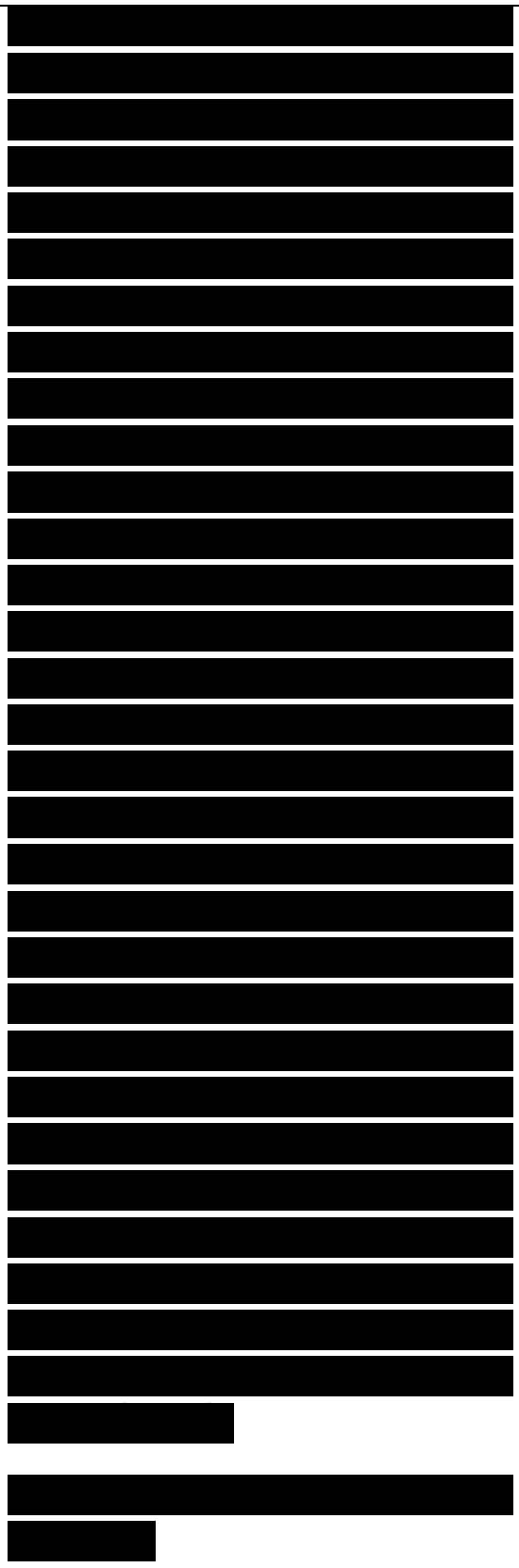
We described a fairly general version of the doubly-connected edge list. In applications where the vertices carry no attribute information we could store their coordinates directly in the Origin() field of the edge;

there is no strict need for a separate type of vertex record. Even more important is to realize that in many applications the faces of the subdivision carry no interesting meaning (think of the network of rivers or roads that we looked at before).

If that is the case, we can completely forget about the face records, and the IncidentFace() field of half-edges. As we will see, the algorithm of the next section doesn't need these fields (and is actually simpler to implement if we don't need to update them). Some implementations of doubly-connected edge lists may also insist that the graph formed by the vertices and edges of the subdivision be connected.

This can always be achieved by introducing dummy edges, and has two advantages. Firstly, a simple graph transversal can be used to visit all half-edges, and secondly, the InnerComponents() list for faces is not necessary.

2.3 Computing the Overlay of Two Subdivisions



Now that we have designed a good representation of a subdivision, we can tackle the general map overlay problem. We define the overlay of two subdivisions §1 and §2 to be the subdivision O(§1, §2) such that there is a face f in O(§1, §2) if and only if there are faces f1 in S1 and f2 in S2 such that f is a maximal connected subset of f1 n f2. This sounds more complicated than it is: what it means is that the overlay is the subdivision of the plane induced by the edges from S1 and S2. Figure 2.4 illustrates this. The general map overlay problem

Figure 2.4

Overlaying two subdivisions is to compute a doubly-connected edge list for O(§1; S2), given the doubly-connected edge lists of S1 and S2. We require that each face in O(§1;S2) be labeled with the labels of the faces in §1 and §2 that contain it. This way we have access to the attribute information stored for these faces. In an overlay of a vegetation map and a precipitation map this would mean that we know for each region in the overlay the type of vegetation and the amount

of precipitation.

Let's first see how much information from the doubly-connected edge lists for S1 and S2 we can re-use in the doubly-connected edge list for O(§1; S2). Consider the network of edges and vertices of §1. This network is cut into pieces by the edges of S2.

These pieces are for a large part re-usable; only the edges that have been cut by the edges of §2 should be renewed. But does this also hold for the half-edge records in the doubly-connected edge list that correspond to the pieces? If the orientation of a half-edge would change, we would still have to change the information in these records. Fortunately, this is not the case. The half-edges are oriented such that the face that they bound lies to the left; the shape of the face may change in the overlay, but it will remain to the same side of the half-edge. Hence, we can re-use half-edge records corresponding to edges that are not intersected by edges from the other map. Stated differently, the only half-edge records in the doubly-connected edge list for O(§1,

S2) that we cannot borrow from §1 or §2 are the ones that are incident to an intersection between edges from different maps.

This suggests the following approach. First, copy the doubly-connected edge lists of §1 and §2 into one new doubly-connected edge list. The new doubly-connected edge list is not a valid doubly-connected edge list, of course, in the sense that it does not yet represent a planar subdivision. This is the task of the overlay algorithm: it must transform the doubly-connected edge list into a valid doubly-connected edge list for O(§1; S2) by computing the intersections between the two networks of edges, and linking together the appropriate parts of the two doubly-connected edge lists.

We did not talk about the new face records yet. The information for these records is more difficult to compute, so we leave this for later. We first describe in a little more detail how the vertex and half-edge records of the doubly- connected edge list for O(§1; S2) are computed.

Our algorithm is based on the plane sweep algorithm of

Section 2.1 for com-puting the intersections in a set of line segments. We run this algorithm on the set of segments that is the union of the sets of edges of the two subdivisions §1 and §2. Here we consider the edges to be closed. Recall that the algorithm is supported by two data structures: an event queue Q, which stores the event points, and the status structure T, which is a balanced binary search tree storing the segments intersecting the sweep line, ordered from left to right. We now also maintain a doubly-connected edge list D. Initially, D contains a copy of the doubly-connected edge list for §1 and a copy of the doubly-connected edge list for S2. During the plane sweep we shall transform D to a correct doubly-connected edge list for O(§1; S2). That is to say, as far as the vertex and half-edge records are concerned; the face information will be computed later.

We keep cross pointers between the edges in the status structure T and the half-edge records in D that correspond to them. This way we can access the part of D

that needs to be changed when we encounter an intersection point. The invariant that we maintain is that at any time during the sweep, the part of the overlay above the sweep line has been computed correctly.

Now, let's consider what we must do when we reach an event point. First of all, we update T and Q as in the line segment intersection algorithm. If the event involves only edges from one of the two subdivisions, this is all; the event point is a vertex that can be re-used.

If the event involves edges from both subdivisions, we must make local changes to D to link the doubly-connected edge lists of the two original subdivisions at the intersection point. This is tedious but not difficult.

Figure 2.5

We describe the details for one of the possible cases, namely when an edge e of S1 passes through a vertex v of S2, see Figure 2.5. The edge e must be replaced by two edges denoted e' and e". In the doubly-connected edge list, the two half-edges for e

must become four. We create two new half-edge records, both with v as the origin. The two existing half-edges for e keep the endpoints of e as their origin, as shown in Figure 2.5. Then we pair up the existing half-edges with the new half-edges by setting their Twin() pointers. So e' is represented by one new and one existing half-edge, and the same holds for e". Now we must set a number of Prev() and Next() pointers. We first deal with the situation around the endpoints of e; later we'll worry about the situation around v. The Next() pointers of the two new half-edges each copy the Next() pointer of the old half-edge that is not its twin. The half-edges to which these pointers point must also update their pointer and set it to the new half-edges. The correctness of this step can be verified best by looking at a figure.

It remains to correct the situation around vertex v. We must set the Next() and Prev() pointers of the four half-edges representing e' and e", and of the four half-edges incident from §2 to v. We locate these four half-edges

from §2 by testing where e' and e" should be in the cyclic order of the edges around vertex v. There are four pairs of half-edges that become linked by a Next() pointer from the one and a Prev() pointer from the other.

Consider the half-edge for e' that has v as its destination. It must be linked to the first half-edge, seen clockwise from e', with v as its origin. The half-edge for e' with v as its origin must be linked to the first counterclockwise half-edge with v as its destination. The same statements hold for e".

Most of the steps in the description above take only constant time. Only locating where e' and e" appear in the cyclic order around v may take longer: it will take time linear in the degree of v. The other cases that can arise—crossings of two edges from different maps, and coinciding vertices—are not more difficult than the case we just discussed. These cases also take time O(m), where m is the number of edges incident to the event

point. This means that updating D does not increase the running time of the line segment intersection algorithm asymptotically.

Notice that every intersection that we find is a vertex of the overlay. It follows that the vertex records and the half-edge records of the doubly-connected edge list for O(§1; S2) can be computed in O(nlogn + klogn) time, where n denotes the sum of the complexities of §1 and §2, and k is the complexity of the overlay.

After the fields involving vertex and half-edge records have been set, it remains to compute the information about the faces of O(§1; §2). More precisely, we have to create a face record for each face f in O(§1; §2), we have to make OuterComponent(f) point to a half-edge on the outer boundary of f, and we have to make a list InnerComponents(f) of pointers to half-edges on the boundaries of the holes inside f. Furthermore, we must set the IncidentFace() fields of the half-edges on the boundary of f so that they point to the face record of f.

Finally, each of the new faces must be labeled with the names of the faces in the old subdivisions that contain it.

How many face records will there be? Well, except for the unbounded face, every face has a unique outer boundary, so the number of face records we have to create is equal to the number of outer boundaries plus one. From the part of the doubly-connected edge list we have constructed so far we can easily extract all boundary cycles. But how do we know whether a cycle is an outer boundary or the boundary of a hole in a face? This can be decided by looking at the leftmost vertex v of the cycle, or, in case of ties, at the lowest of the leftmost ones. Recall that half-edges are directed in such a way that their incident face locally lies to the left. Consider the two half-edges of the cycle that are incident to v. Because we know that the incident face lies to the left, we can compute the angle these two half-edges make inside the incident face. If this angle is smaller than 180° then the cycle is an outer boundary, and otherwise it is the

boundary of a hole. This property holds for the leftmost vertex of a cycle, but not necessarily for other vertices of that cycle.

To decide which boundary cycles bound the same face we construct a graph s. For every boundary cycle—inner and outer—there is a node in s. There is also one node for the imaginary outer boundary of the unbounded face. There is an arc between two cycles if and only if one of the cycles is the boundary of a hole and the other cycle has a half-edge immediately to the left of the leftmost vertex of that hole cycle. If there is no half-edge to the left of the leftmost vertex of a cycle, then the node representing the cycle is linked to the node of the unbounded face. Figure 2.6 gives an example. The dotted segments in the figure indicate the linking of the hole cycles to other cycles. The graph corresponding to the subdivision is also shown in the figure. The hole cycles are shown as single circles, and the outer boundary cycles are shown as double circles. Observe that C3 and C6 are in the same connected component as C2. This indicates that C3 and C6 are hole cycles in the face whose

outer boundary is C2. If there is only one hole in a face f, then the graph s links the boundary cycle of the hole to the outer boundary of f. In general this need not be the case: a hole can also be linked to another hole, as you can see in Figure 2.6. This hole, which lies in the same face f, may be linked to the outer boundary of f, or it may be linked to yet another hole. But eventually we must end up linking a hole to the outer boundary, as the next lemma shows.

Lemma 2.5 Each connected component of the graph s corresponds exactly to the set of cycles incident to one face.

Proof. Consider a cycle C bounding a hole in a face f. Because f lies locally to the left of the leftmost vertex of C, C must be linked to another cycle that also

Figure 2.6
A subdivision and the corresponding graph s

bounds f. It follows that cycles in the same connected component of s bound the same face.

To finish the proof, we show that every cycle bounding a

hole in f is in the same connected component as the outer boundary of f. Suppose there is a cycle for which this is not the case. Let 6 be the leftmost such cycle, that is, the one whose the leftmost vertex is leftmost. By definition there is an arc between the 6 and another cycle 6' that lies partly to the left of the leftmost vertex of 6. Hence, 6' is in the same connected component as 6, which is not the component of the outer boundary of f. This contradicts the definition of 6. 0

Lemma 2.5 shows that once we have the graph s, we can create a face record for every component. Then we can set the IncidentFace() pointers of the halfedges that bound each face f, and we can construct the list InnerComponents(f) and the set OuterComponent(f). How can we construct s?

Recall that in the plane sweep algorithm for line segment intersection we always looked for the segments immediately to the left of an event point.

(They had to be tested for intersection against the

leftmost edge through the event point.) Hence, the information we need to construct G is determined during the plane sweep. So, to construct s, we first make a node for every cycle. To find the arcs of s, we consider the leftmost vertex v of every cycle bounding a hole. If e is the half-edge immediately left of v, then we add an arc between the two nodes in s representing the cycle containing e and the hole cycle of which v is the leftmost vertex. To find these nodes in s efficiently we need pointers from every half-edge record to the node in s representing the cycle it is in. So the face information of the doubly-connected edge list can be set in O(n + k) additional time, after the plane sweep.

One thing remains: each face f in the overlay must be labeled with the names of the faces in the old subdivisions that contained it. To find these faces, consider an arbitrary vertex v of f. If v is the intersection of an edge e1 from S1 and an edge e2 from S2 then we can decide which faces of S1 and S2 contain f by looking at the

IncidentFace() pointer of the appropriate half-edges corresponding to e1 and e2. If v is not an intersection but a vertex of, say, S1, then we only know the face of S1 containing f. To find the face of S2 containing f, we have to do some more work: we have to determine the face of S2 that contains v. In other words, if we knew for each vertex of S1 in which face of S2 it lay, and vice versa, then we could label the faces of O(§1, S2) correctly. How can we compute this information? The solution is to apply the paradigm that has been introduced in this chapter, plane sweep, once more. However, we won't explain this final step here. It is a good exercise to test your understanding of the plane sweep approach to design the algorithm yourself. (In fact, it is not necessary to compute this information in a separate plane sweep. It can also be done in the sweep that computes the intersections.)

Putting everything together we get the following algorithm.

Algorithm MAPOVERLAY(§1, S2)

Input. Two planar subdivisions S1 and S2 stored in doubly-connected edge lists. Output. The overlay of S1 and S2 stored in a doubly-connected edge list D.

1. Copy the doubly-connected edge lists for S1 and S2 to a new doubly-connected edge list D.

2. Compute all intersections between edges from S1 and S2 with the plane sweep algorithm of Section 2.1. In addition to the actions on T and Q required at the event points, do the following:

- Update D as explained above if the event involves edges of both S1 and S2. (This was explained for the case where an edge of S1 passes through a vertex of S2.)
- Store the half-edge immediately to the left of the event point at the vertex in D representing it.

(* Now D is the doubly-connected edge list for O(§1, S2), except that the information about the faces has not been computed yet. *)

Determine the boundary cycles in O(§1, S2) by traversing D. Construct the graph s whose nodes correspond to boundary cycles and whose arcs connect each hole cycle to the cycle to the left of its leftmost ver-tex, and compute its connected components. (The information to determine the arcs of s has been computed in line 2, second item.) for each connected component in s do Let C be the unique outer boundary cycle in the component and let f denote the face bounded by the cycle. Create a face record for f, set OuterComponent(f) to some half-edge of C, and construct the list InnerComponents(f) consisting of pointers to one half-edge in each hole cycle in the component. Let the IncidentFace() pointers of all half-edges in the cycles point to the face record of f. 8. Label each face of O(S1; S2) with the names of the faces of S1 and S2 containing it, as explained above. Theorem 2.6 Let S1 be a planar subdivision of	[redacted]	

complexity n1, let S2 be a subdivision of complexity n2, and let n := n1 + n2. The overlay of S1 and S2 can be constructed in O(n log n + klog n) time, where k is the complexity of the overlay.

Proof. Copying the doubly-connected edge lists in line 1 takes O(n) time, and the plane sweep of line 2 takes O(nlog n + klog n) time by Lemma 2.3. Steps 4-7, where we fill in the face records, takes time linear in the complexity of O(S1; S2). (The connected components of a graph can be determined in linear time by a simple depth first search.) Finally, labeling each face in the resulting subdivision with the faces of the original subdivisions that contain it can be done in O(nlogn + klogn) time.

2.4 Boolean Operations

The map overlay algorithm is a powerful instrument that can be used for various other applications. One particular useful one is performing the Boolean operations union, intersection, and difference on two polygons P1 and P2. See Figure 2.7 for an

example. Note that the output of the operations might no longer be a polygon. It can consist of a number of polygonal regions, some with holes.

Figure 2.7

The Boolean operations union, intersection and difference on two polygons Pi and P2

To perform the Boolean operation we regard the polygons as planar maps whose bounded faces are labeled P1 and P2, respectively. We compute the overlay of these maps, and we extract the faces in the overlay whose labels correspond to the particular Boolean operation we want to perform. If we want to compute the intersection P1 n P2, we extract the faces in the overlay that are labeled with P1 and P2.

If we want to compute the union P1 u P2, we extract the faces in the overlay that are labeled with P1 or P2. And if we want to compute the difference P1 \ P2, we extract the faces in the overlay that are labeled with P1 and not with P2.

Because every intersection point of an edge of P1 and an edge of P2 is a vertex of P1 n P2, the running time of the

algorithm is O(nlogn + klogn), where n is the total number of vertices in P1 and P2, and k is the complexity of P1 n P2. The same holds for the other Boolean operations: every intersection of two edges is a vertex of the final result, no matter which operation we want to perform. We immediately get the following result.

Corollary 2.7 Let P1 be a polygon with n1 vertices and P2 a polygon with n2 vertices, and let n := n1 + n2. Then P1 n P2, P1 u P2, and P1 \ P2 can each be computed in O(n log n + k log n) time, where k is the complexity of the output.

2.5 Notes and Comments

The line segment intersection problem is one of the most fundamental problems in computational geometry. The O(nlog n + klog n) solution presented in this chapter was given by Bentley and Ottmann [47] in 1979. (A few years earlier, Shamos and Hoey [351] had solved the detection problem, where one is only interested in deciding whether there is at least one intersection, in O(nlogn) time.) The method for reducing the working storage from O(n + k) to O(n)

described in this chapter is taken from Pach and Sharir [312], who also show that the event list can have size Q.(nlogn) before this improvement. Brown [77] describes an alternative method to achieve the reduction.

The lower bound for the problem of reporting all line segment intersections is Q.(nlogn + k), so the plane sweep algorithm described in this chapter is not optimal when k is large. A first step towards an optimal algorithm was taken by Chazelle [88], who gave an algorithm with O(nlog2n/loglogn + k) running time. In 1988 Chazelle and Edelsbrunner [99, 100] presented the first O(nlog n + k) time algorithm. Unfortunately, it requires O(n + k) storage. Later Clarkson and Shor [133] and Mulmuley [288] gave randomized incremental algorithms whose expected running time is also O(nlogn + k). (See Chapter 4 for an explanation of randomized algorithms.) The working storage of these algorithms is O(n) and O(n + k), respectively. Unlike the algorithm of Chazelle and Edelsbrunner, these randomized algorithms also

work for computing intersections in a set of curves. Balaban [35] gave the first deterministic algorithm for the segment intersection problem that works in O(nlog n + k) time and O(n) space. It also works for curves.

There are cases of the line segment intersection problem that are easier than the general case. One such case is where we have two sets of segments, say red segments and blue segments, such that no two segments from the same set intersect each other. (This is, in fact, exactly the network overlay problem. In the solution described in this chapter, however, the fact that the segments came from two sets of non-intersecting segments was not used.) This so-called red-blue line segment intersection problem was solved in O(nlog n + k) time and O(n) storage by Mairson and Stolfi [262] before the general problem was solved optimally. Other optimal red-blue intersection algorithms were given by Chazelle et al. [101] and by Palazzi and Snoeyink [315]. If the two sets of segments form connected subdivisions then the situation is even

better: in this case the overlay can be computed in O(n + k) time, as has been shown by Finke and Hinrichs [176]. Their result generalizes and improves previous results on map overlay by Nievergelt and Preparata [293], Guibas and Seidel [200], and Mairson and Stolfi [262]. The line segment intersection counting problem is to determine the number of intersection points in a set of n line segments. Since the output is a single integer, a term with k in the time bound no longer refers to the output size (which is constant), but only to the number of intersections. Algorithms that do not depend on the number of intersections take O(n4/3 logcn) time, for some small constant c [4, 95]; a running time close to O(nlogn) is not known to exist. Plane sweep is one of the most important paradigms for designing geometric algorithms. The first algorithms in computational geometry based on this paradigm are by Shamos and Hoey [351], Lee and Preparata [250], and Bentley and Ottmann [47]. Plane		

sweep algorithms are especially suited for finding intersections in sets of objects, but they can also be used for solving many other problems. In Chapter 3 plane sweep solves part of the polygon triangulation problem, and in Chapter 7 we will see a plane sweep algorithm to compute the so-called Voronoi diagram of a set of points. The algorithm presented in the current chapter sweeps a horizontal line downwards over the plane. For some problems it is more convenient to sweep the plane in another way. For instance, we can sweep the plane with a rotating line—see Chapter 15 for an example—or with a pseudo-line (a line that need not be straight, but otherwise behaves more or less as a line) [159]. The plane sweep technique can also be used in higher dimensions: here we sweep the space with a hyperplane [213, 311, 324]. Such algorithms are called space sweep algorithms. In this chapter we described a data structure for storing subdivisions: the doubly-connected edge list. This structure, or in fact a variant		

of it, was described by Muller and Preparata [286]. There are also other data structures for storing subdivisions, such as the winged edge structure by Baumgart [40] and the quad edge structure by Guibas and Stolfi [202]. The difference between all these structures is small. They all have more or less the same functionality, but some save a few bytes of storage per edge.	[redacted]	

7 Voronoi Diagrams The Post Office Problem Suppose you are on the advisory board for the planning of a supermarket chain, and there are plans to open a new branch at a certain location. To predict whether the new branch will be profitable, you must estimate the number of customers it will attract. For this you have to model the behavior of your potential customers: how do people decide where to do their shopping? A similar question arises in social geography, when studying the economic activities in a country: what is the trading area of certain cities? In a more abstract setting we have a set of Figure 7.1 The trading areas of the capitals of the twelve provinces in the Netherlands, as predicted by the Voronoi assignment model central places—called sites—that provide certain goods or services, and we want to know for each site where the people live who obtain their goods or services from that site. (In computational	7 Sơ đồ Voronoi Bài toán bưu điện Giả sử bạn ở trong hội đồng tư vấn cho một dự án xây dựng hệ thống siêu thị, và có kế hoạch mở một chi nhánh mới tại một địa điểm nào đó. Để dự đoán được chi nhánh mới này có thu được lợi nhuận hay không, bạn phải ước tính được số khách hàng mà nó có thể thu hút được. Để làm được điều này, bạn phải mô hình hóa hành vi của các khách hàng tiềm năng của bạn: cách thức họ chọn nơi mua sắm? Một câu hỏi tương tự cũng phát sinh trong địa lý xã hội, khi nghiên cứu các hoạt động kinh tế ở một quốc gia: Khu vực nào là khu vực thương mại của thành phố? Trong một cấu hình trừu tượng hơn, chúng ta có một tập hợp các Hình 7.1 Các khu vực thương mại của các thủ phủ tương ứng với 12 tỉnh ở Hà Lan, theo dự đoán của mô hình phân định Voronoi địa điểm trung tâm-được gọi là các vị trí-cung cấp cho chúng ta thông tin về hàng hóa và dịch vụ, và chúng ta cũng cần biết ở những vị trí đó ai là người chấp nhận mua hàng hóa và dịch vụ tại chỗ (Trong	

geometry the sites are traditionally viewed as post offices where customers want to post their letters—hence the subtitle of this chapter.) To study this question we make the following simplifying assumptions: ■ the price of a particular good or service is the same at every site; ■ the cost of acquiring the good or service is equal to the price plus the cost of transportation to the site; 147 ■ the cost of transportation to a site equals the Euclidean distance to the site times a fixed price per unit distance; ■ consumers try to minimize the cost of acquiring the good or service. Usually these assumptions are not completely satisfied: goods may be cheaper at some sites than at others, and the transportation cost between two points is probably not linear in the Euclidean distance between them. But the model above can give a rough approximation of the trading areas of the sites. Areas where the behavior of the people differs from that predicted by the model can be subjected to further research,	hình học tính toán, thông thường các vị trí được xem như các bưu điện ở đó khách hàng cần gửi thư của họ, đó cũng là tiêu đề của chương này.) Để nghiên cứu những vấn đề này, chúng ta cần phải đặt ra các giả thuyết như sau: ■ giá của một hàng hóa hay dịch vụ như nhau ở mọi vị trí; ■ chi phí mua hàng hóa hay dịch vụ bằng giá cộng với chi phí vận chuyển đến các vị trí đó; 147 ■ chi phí vận chuyển đến một vị trí bằng khoảng cách Euclide đến vị trí đó nhân giá cố định trên một đơn vị khoảng cách; ■ người tiêu dùng cố gắng giảm thiểu chi phí mua hàng hóa hay dịch vụ.Thông thường những giả thuyết này không được thỏa mãn hoàn toàn: hàng hóa ở một số vị trí có thể rẻ hơn ở một số vị trí khác, và chi phí vận chuyển giữa hai điểm có thể không phụ thuộc tuyến tính vàokhoảng cách Euclide giữa chúng. Nhưng mô hình ở trên cho chúng ta thấy một gần đúng thô về khu vực thương mại gồm nhiều vị trí. Khu vực mà trong đó con người hành động không đúng với những gì mô hình tiên đoán cần phải được nghiên cứu thêm, để xem điều gì đã	

to see what caused the different behavior. Our interest lies in the geometric interpretation of the model above. The assump-tions in the model induce a subdivision of the total area under consideration into regions—the trading areas of the sites—such that the people who live in the same region all go to the same site. Our assumptions imply that people simply get their goods at the nearest site—a fairly realistic situation. This means that the trading area for a given site consists of all those points for which that site is closer than any other site. Figure 7.1 gives an example. The sites in this figure are the capitals of the twelve provinces in the Netherlands. The model where every point is assigned to the nearest site is called the Voronoi assignment model. The subdivision induced by this model is called the Voronoi diagram of the set of sites. From the Voronoi diagram we can derive all kinds of information about the trading areas of the sites and their relations. For example, if the regions of two sites have a common boundary then these two sites are likely to be in	gây ra những khác biệt đó. Ở đây, chúng ta chỉ quan tâm đến khía cạnh hình học của mô hình nói trên. Các giả thiết trong mô hình dẫn đến việc cần phải chia nhỏ khu vực đang xét thành các vùng-các khu vực thương mại bao gồm các vị trí-sao cho tất cả những người sống trong cùng một vùng đều đi đến cùng một vị trí. Giả thuyết của chúng ta chỉ đơn giản là, người ta chỉ mua hàng hóa ở vị trí gần nhất-một tình huống rất phù hợp với thực tế. Điều này có nghĩa là khu vực thương mại đối với một vị trí nhất định bao gồm tất cả những điểm nào gần với vị trí đó hơn so với bất kỳ vị trí nào khác. Xem ví dụ trong Hình 7.1. Các vị trí trong hình này là các thủ phủ của 12 tỉnh ở Hà Lan. Mô hình trong đó mỗi điểm được ấn định với vị trí gần nhất được gọi là mô hình phân định Voronoi. Việc phân chia của mô hình này được gọi là sơ đồ Voronoi của tập hợp các vị trí. Từ sơ đồ Voronoi, chúng ta có thể rút ra được tất cả các loại thông tin về khu vực thương mại của các vị trí và mối quan hệ giữa chúng. Ví dụ, nếu các vùng của hai vị trí có biên chung thì hai vị trí này có lẽ ở trong tình	

direct competition for customers that live in the boundary region. The Voronoi diagram is a versatile geometric structure. We have described an application to social geography, but the Voronoi diagram has applications in physics, astronomy, robotics, and many more fields. It is also closely linked to another important geometric structure, the so-called Delaunay triangulation, which we shall encounter in Chapter 9. In the current chapter we shall confine ourselves to the basic properties and the construction of the Voronoi diagram of a set of point sites in the plane. Definition and Basic Properties Denote the Euclidean distance between two points p and q by dist(p, q). In the plane we have dist(p, q) :=\J(Px - qx)2 + (Py - qy)2. Let P: = {p1, p2pn} be a set of n distinct points in the plane; these points are the sites. We define the Voronoi diagram of P as the subdivision of the plane into n cells,	trạng cạnh tranh trực tiếp để lôi kéo khách hàng sống trong vùng biên. Sơ đồ Voronoi là một cấu trúc hình học đa năng. Chúng ta đã mô tả một ứng dụng cho địa lý xã hội, nhưng các sơ đồ Voronoi cũng có những ứng dụng trong vật lý, thiên văn học, robot học, và nhiều lĩnh vực khác. Nó cũng có mối liên hệ mật thiết với một cấu trúc hình học quan trọng khác, được gọi là lưới tam giác Delaunay, một vấn đề mà chúng ta sẽ xét trong Chương 9. Trong chương này, chúng ta chỉ tập trung vào các tính chất cơ bản và cấu trúc của sơ đồ Voronoi của tập hợp các vị trí điểm trong mặt phẳng. Định nghĩa và các tính chất cơ bản Kí hiệu khoảng cách Euclide giữa hai điểm p và q là…. (p, q). Trong mặt phẳng, chúng ta có Quận (p, q): = \ J (px - qx) 2 + (Py - qy) 2. Đặt P = {p1, p2pn} là một tập hợp các n điểm phân biệt trong mặt phẳng, những điểm này là các vị trí. Chúng ta định nghĩa sơ đồ Voronoi của P là sự chia nhỏ mặt phẳng thành n ô,	

9 Delaunay Triangulations
Height Interpolation 10/8

When we talked about maps of a piece of the earth's surface in previous chapters, we implicitly assumed there is no relief. This may be reasonable for a country like the Netherlands, but it is a bad assumption for Switzerland. In this chapter we set out to remedy this situation.

We can model a piece of the earth's surface as a terrain. A terrain is a 2-dimensional surface in 3-dimensional space with a special property: every vertical line intersects it in a point, if it intersects it at all. In other words, it is the graph of a function f: A c R2 ^ R that assigns a height f (p) to every point p in the domain, A, of the terrain. (The earth is round, so on a global scale terrains defined in this manner are not a good model of the earth. But on a more local scale terrains provide a fairly good model.) A terrain can be visualized with a perspective drawing like the one in Figure 9.1, or with contour lines—lines of equal height—like on a topographic map.

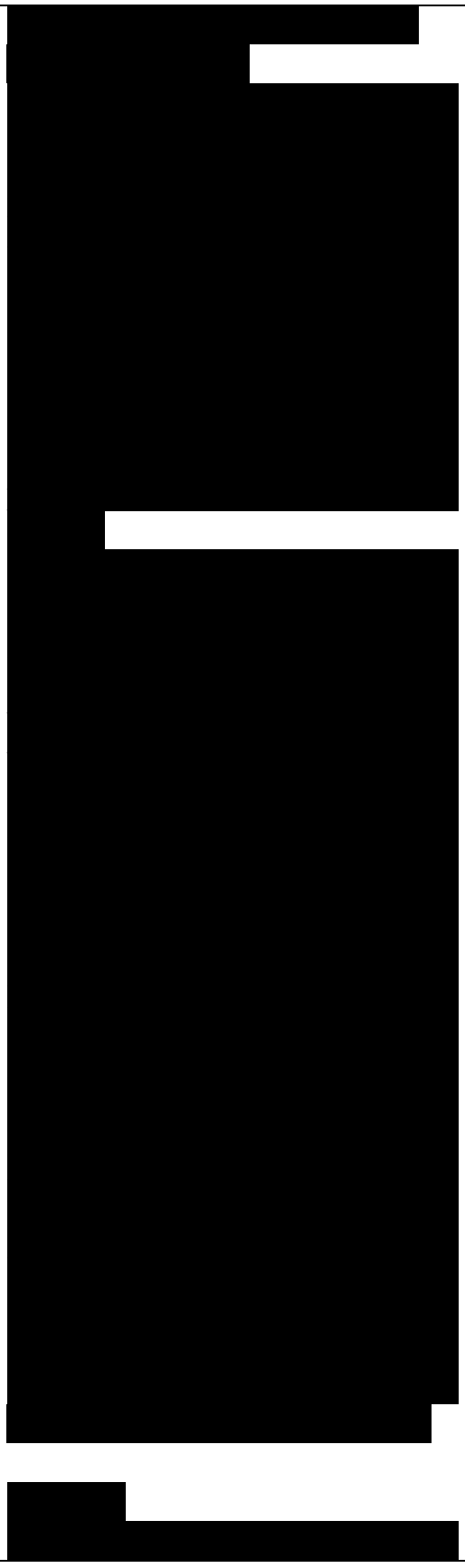


Figure 9.1
A perspective view of a terrain

Of course, we don't know the height of every point on earth; we only know it where we've measured it. This means that when we talk about some terrain, we only know the value of the function f at a finite set P c A of sample points. From the height of the sample points we somehow have to approximate the height at the other points in the domain. A naive approach assigns to every p e A the height of the nearest sample point. However, this gives a discrete terrain, which doesn't look very natural. Therefore our approach for approximating a terrain is as follows. We first determine a triangulation of P: a planar subdivision whose bounded faces are triangles and whose vertices are the points of P. (We assume that the sample points are such that we can make the triangles cover the domain of the terrain.) We then lift each sample point to its correct height, thereby mapping every triangle in the triangulation to a triangle in 3-space. Figure 9.2 illustrates this. What we get is a polyhedral terrain, the graph of a continuous function that is piecewise linear. We can use the polyhedral terrain as an approximation of the original terrain.		

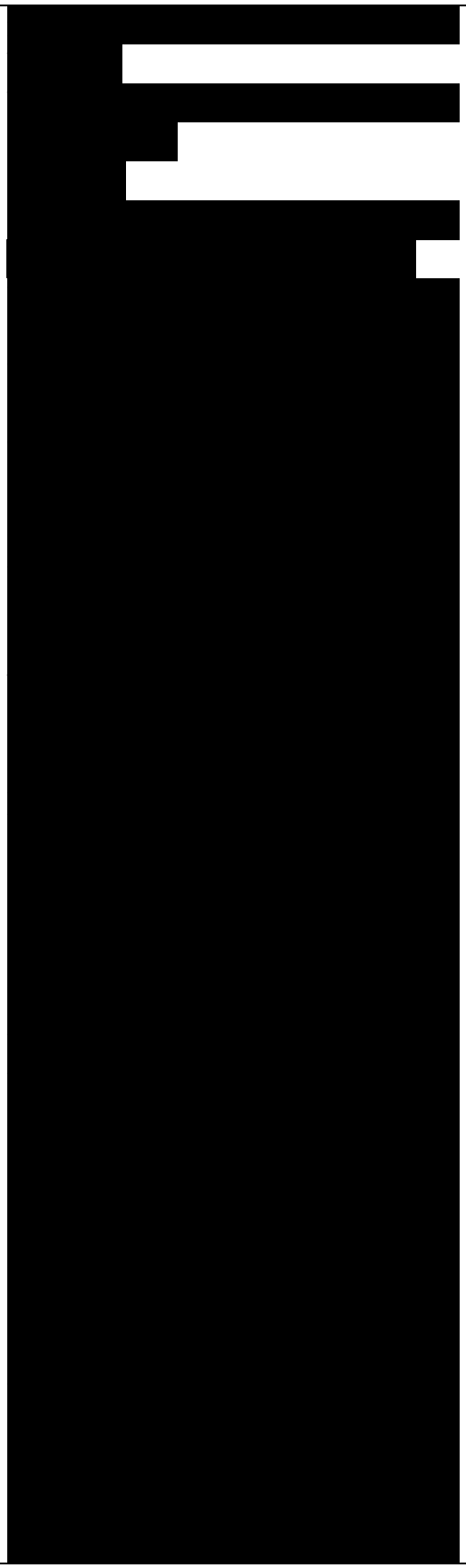
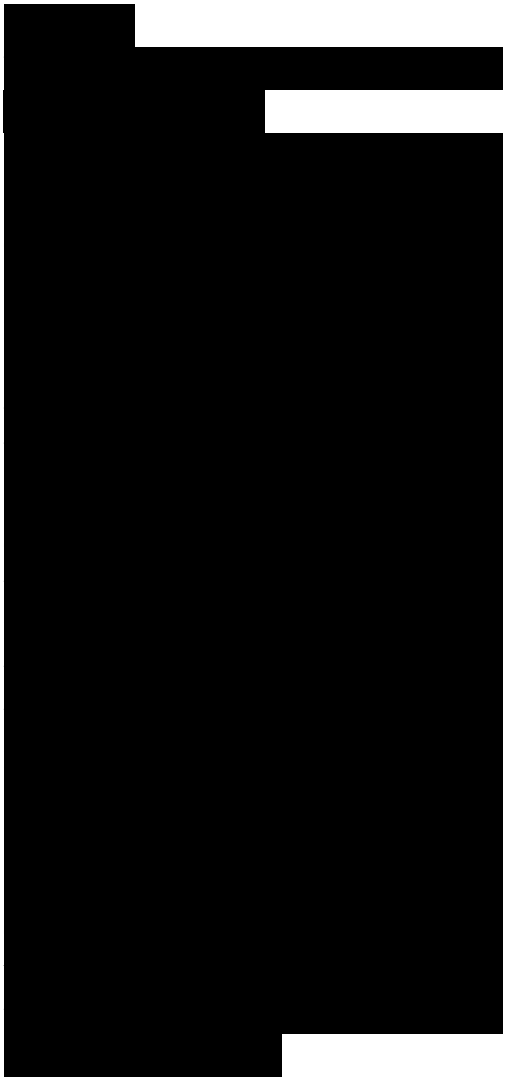


Figure 9.2
Obtaining a polyhedral terrain from a set of sample points

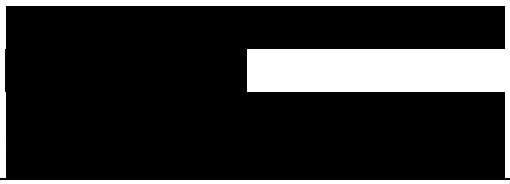
The question remains: how do we triangulate the set of sample points? In general, this can be done in many different ways. But which triangulation is the most appropriate one for our purpose, namely to approximate a terrain? There is no definitive answer to this question. We do not know the original terrain, we only know its height at the sample points. Since we have no other information, and the height at the sample points is the correct height for any triangulation, all triangulations of P seem equally good. Nevertheless, some triangulations look more natural than others. For example, have a look at Figure 9.3, which shows two triangulations of the same point set. From the heights of the sample points we get the impression that the sample points were taken from a mountain ridge. Triangulation (a) reflects this intuition. Triangulation (b), however, where one single edge has been "flipped," has introduced a narrow valley cutting through

the mountain ridge. Intuitively, this looks wrong. Can we turn this intuition into a criterion that tells us that triangulation (a) is better than triangulation (b)?

Figure 9.3
Flipping one edge can make a big difference



The problem with triangulation (b) is that the height of the point q is determined by two points that are relatively far away. This happens because q lies in the middle of an edge of two long and sharp triangles. The skinniness of these triangles causes the trouble. So it seems that a triangulation that contains small angles is bad. Therefore we will rank triangulations by comparing their smallest angle. If the minimum angles of two triangulations are identical, then we can look at the second smallest angle, and so on. Since there is only a finite number of different triangulations of a given point set P, this implies that there must be an optimal triangulation, one that maximizes the minimum angle. This will be the triangulation we are looking for.



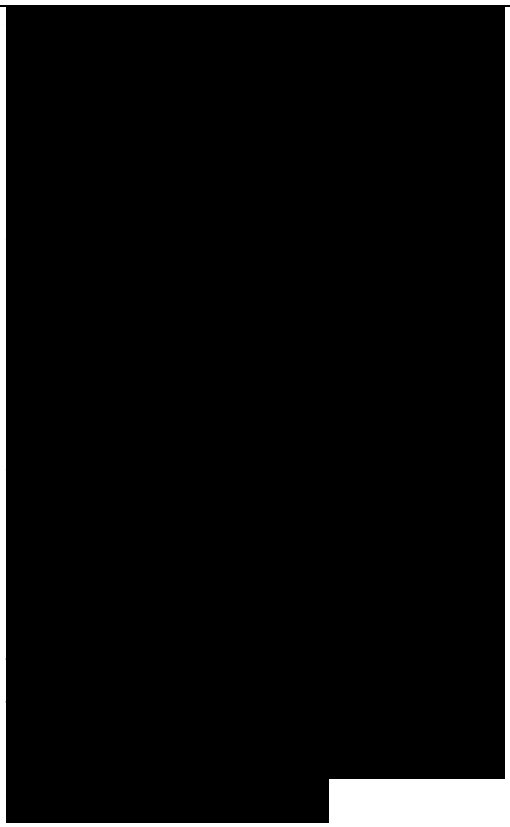
9.1 Triangulations of Planar Point Sets

Let P := {p1, p2,..., pn} be a set of points in the plane. To be

able to formally define a triangulation of P, we first define a maximal planar subdivision as a subdivision S such that no edge connecting two vertices can be added to S without destroying its planarity. In other words, any edge that is not in S intersects one of the existing edges. A triangulation of P is now defined as a maximal planar subdivision whose vertex set is P.

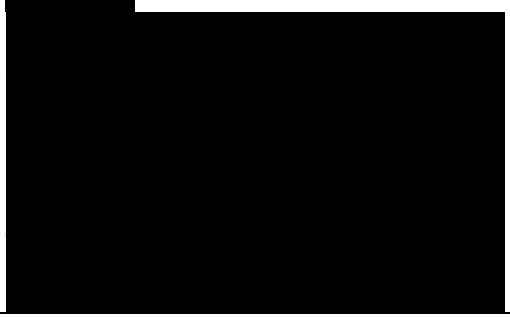
With this definition it is obvious that a triangulation exists. But does it consist of triangles? Yes, every face except the unbounded one must be a triangle: a bounded face is a polygon, and we have seen in Chapter 3 that any polygon can be triangulated. What about the unbounded face? It is not difficult to see that any segment connecting two consecutive points on the boundary of the convex hull of P is an edge in any triangulation T. This implies that the union of the bounded faces of T is always the convex hull of P, and that the unbounded face is always the

complement of the convex hull. (In our application this means that if the domain is a rectangular area, say, we have to make sure that the corners of the domain are included in the set of sample points, so that the triangles in the triangulation cover the domain of the terrain.) The number of triangles is the same in any triangulation of P. This also holds for the number of edges. The exact numbers depend on the number of points in P that are on the boundary of the convex hull of P. (Here we also count points in the interior of convex hull edges. Hence, the number of points on the convex hull boundary is not necessarily the same as the number of convex hull vertices.) This is made precise in the following theorem.



Theorem 9.1 Let P be a set of n points in the plane, not all collinear, and let k denote the number of points in P that lie on the boundary of the convex hull of P. Then any triangulation of P has 2n - 2 - k triangles and 3n - 3 - k edges.

Proof. Let T be a triangulation of P, and let m denote the number of triangles of T. Note that the number of faces of the triangulation, which we denote by nf, is m + 1. Every triangle has three edges, and the



unbounded face has k edges. Furthermore, every edge is incident to exactly two faces. Hence, the total number of edges of T is ne := (3m + k)/2. Euler's formula tells us that
n - ne + nf = 2.

Plugging the values for ne and nf into the formula, we get m = 2n - 2 - k, which in turn implies ne = 3n - 3 - k. 0
Let T be a triangulation of P, and suppose it has m triangles. Consider the 3m angles of the triangles of T, sorted by increasing value. Let ai, a2a3m be the resulting sequence of angles; hence, ai < aj, for i < j. We call A(T) := (a1, a2,..., a3m) the angle-vector of T. Let T' be another triangulation of the same point set P, and let A(T') := (ai, a2,..., a3m) be its angle-vector. We say that the angle-vector of T is larger than the angle-vector of T' if A(T) is lexicographically larger than A(T'), or, in other words, if there exists an index i with 1 < i < 3m such that
aj = aj for all j < i, and ai > aj.

We denote this as A(T) > A(T'). A triangulation T is called angle-optimal if A(T) >

A(T') for all triangulations Tj of P. Angle-optimal triangulations are interesting because, as we have seen in the introduction to this chapter, they are good triangulations if we want to construct a polyhedral terrain from a set of sample points.

Below we will study when a triangulation is angle-optimal. To do this it is useful to know the following theorem, often called Thales's Theorem. Denote the smaller angle defined by three points p, q, r by Åpqr.

Theorem 9.2 Let C be a circle, I a line intersecting C in points a and b, and p, q, r, and s points lying on the same side of I. Suppose that p and q lie on C, that r lies inside C, and that s lies outside C. Then
Åarb > Åapb = Åaqb > Åasb.

Now consider an edge e = pip] of a triangulation T of P. If e is not an edge of the unbounded face, it is incident to two triangles pipjpk and pipjpi. If these two triangles form a convex quadrilateral, we can obtain a new triangulation Tj by removing pipj from T and inserting pkpi instead. We call

this operation an edge flip. The only difference in the angle-vector of T and Tj are the six angles a1,...,a6 in A(T), which are replaced by ai ,...,a6 in A(T'). Figure 9.4 illustrates this. We call the edge e = pTpj an illegal edge if
min ai < min aj.
1<i<6 1<i<6 ‘

In other words, an edge is illegal if we can locally increase the smallest angle by flipping that edge. The following observation immediately follows from the definition of an illegal edge.

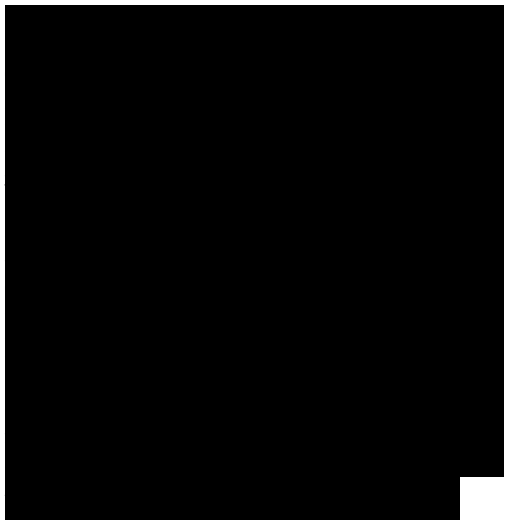
Observation 9.3 Let T be a triangulation with an illegal edge e. Let T' be the triangulation obtained from T by flipping e. Then A(T') > A(T).

It turns out that it is not necessary to compute the angles a1,...,a6, a[,...,a6 to check whether a given edge is legal. Instead, we can use the simple criterion stated in the next lemma. The correctness of this criterion follows from Thales’s Theorem.

Lemma 9.4 Let edge pipj be incident to triangles pipjpk and pipjpl, and let C be the circle through pi, pj, and pk. The

edge pTpj is illegal if and only if the point pl lies in the interior of C. Furthermore, if the points pi, pj, pk, pl form a convex quadrilateral and do not lie on a common circle, then exactly one of pTpj and pkpl is an illegal edge.

Observe that the criterion is symmetric in pk and pl: pl lies inside the circle through pi, pj, pk if and only if pk lies inside the circle through pi, pj, pl. When all four points lie on a circle, both pipj and pkpi are legal. Note that the two triangles incident to an illegal edge must form a convex quadrilateral, so that it is always possible to flip an illegal edge.



We define a legal triangulation to be a triangulation that does not contain any illegal edge. From the observation above it follows that any angle-optimal triangulation is legal. Computing a legal triangulation is quite simple, once we are given an initial triangulation. We simply flip illegal edges until all edges are legal.

Algorithm LEGALTRIANGULATION(T)

Input. Some triangulation T of a point set P.

Output. A legal triangulation of P.
1. while T contains an illegal edge pipj
2. do (* Flip pTpj *)
3. Let pipjpk and pipjpl be the two triangles adjacent to pipj.
4. Remove pipj from T, and add pkpl instead.
5. return T

Why does this algorithm terminate? It follows from Observation 9.3 that the angle-vector of T increases in every iteration of the loop. Since there is only a finite number of different triangulations of P, this proves termination of the algorithm. Once it terminates, the result is a legal triangulation. Although the algorithm is guaranteed to terminate, it is too slow to be interesting. We have given the algorithm anyway, because later we shall need a similar procedure.
But first we will look at something completely different—or so it seems. 195

9.2 The Delaunay Triangulation

Let P be a set of n points—or sites, as we shall sometimes call them—in the plane. Recall from Chapter 7 that the Voronoi diagram of P is the

subdivision of the plane into n regions, one for each site in P, such that the region of a site p e P contains all points in the plane for which p is the closest site. The Voronoi diagram of P is denoted by Vor(P). The region of a site p is called

Figure 9.5 The dual graph of Vor(P)

the Voronoi cell of p; it is denoted by V(p). In this section we will study the dual graph of the Voronoi diagram. This graph g has a node for every Voronoi cell—equivalently, for every site—and it has an arc between two nodes if the corresponding cells share an edge. Note that this means that g has an arc for every edge of Vor(P). As you can see in Figure 9.5, there is a one-to-one correspondence between the bounded faces of g and the vertices of Vor(P).

Figure 9.6 The Delaunay graph Dg(P)

Consider the straight-line embedding of g, where the node corresponding to the Voronoi cell V(p) is the point p, and the arc connecting the nodes of V(p) and V(q) is the segment pq—see Figure 9.6. We call this embedding the Delaunay graph of P, and we denote it by Dg(P). (Although the name sounds French,

Delaunay graphs have nothing to do with the French painter. They are named after the Russian mathematician Boris Nikolaevich Delone, who wrote his own name as “BopHc HHKoiaeBHq DeioHe,” which would be transliterated into English as “Delone.” However, since his work was published in French—at his time, the languages of science were French and German—his name is better known in the French transliteration.) The Delaunay graph of a point set turns out to have a number of surprising properties. The first is that it is always a plane graph: no two edges in the embedding cross.

Theorem 9.5 The Delaunay graph of a planar point set is a plane graph.

Proof. To prove this, we need a property of the edges in the Voronoi diagram stated in Theorem 7.4(ii). For completeness we repeat the property, phrased here in terms of Delaunay graphs.

The edge pipj is in the Delaunay graph Dg(P) if and only if there is a closed disc Cij with pi and pj on its boundary

and no other site of P contained in it. (The center of such a disc lies on the common edge of V(pi) and v(pj).)
Define tij to be the triangle whose vertices are pi, pj, and the center of Cij. Note that the edge of tij connecting pi to the center of Cij is contained in V(pi); a similar observation holds for pj. Now let pkpl be another edge of Dg(P), and define the circle Ckl and the triangle tkl similar to the way Cij and tij were defined.

Suppose for a contradiction that pipj and pkpl intersect. Both pk and pl must lie outside Cij and so they also lie outside tij. This implies that pkpl must intersect one of the edges of tij incident to the center of Cij. Similarly, pTpj must intersect one of the edges of tkl incident to the center of Ckl. It follows that one of the edges of tij incident to the center of Cij must intersect one of the edges of tkl incident to the center of Ckl. But this contradicts that these edges are contained in disjoint Voronoi cells.

The Delaunay graph of P is an embedding of the dual graph of the Voronoi diagram. As observed earlier, it has a face for every vertex of Vor(P). The edges around a face correspond

to the Voronoi edges incident to the corresponding Voronoi vertex. In particular, if a vertex v of Vor(P) is a vertex of the Voronoi cells for the sites p1; p2, p3,..., pk, then the corresponding face f in Dg(P) has p1,p2, p3,...,pk as its vertices. Theorem 7.4(i) tells us that in this situation the points p1;p2,p3,...,pk lie on a circle around v, so we not only know that f is a k-gon, but even that it is convex.

If the points of P are distributed at random, the chance that four points happen to lie on a circle is very small. We will—in this chapter—say that a set of points is in general position if it contains no four points on a circle. If P is in general position, then all vertices of the Voronoi diagram have degree three, and consequently all bounded faces of Dg(P) are triangles. This explains why DG(P) is often called the Delaunay triangulation of P. We shall be a bit more careful, and will call Dg(P) the Delaunay graph of P. We define a Delaunay triangulation to be any triangulation obtained by adding edges to the Delaunay graph. Since all faces of Dg(P) are convex, obtaining such a triangulation is easy. Observe that the Delaunay triangulation of P is unique if

and only if DG(P) is a triangulation, which is the case if P is in general position.

We now rephrase Theorem 7.4 about Voronoi diagrams in terms of Delaunay graphs.

Theorem 9.6 Let P be a set of points in the plane.

(i) Three points pi, pj, pk e P are vertices of the same face of the Delaunay graph of P if and only if the circle through pi, pj, pk contains no point of P in its interior.

(ii) Two points pi, pj e P form an edge of the Delaunay graph of P if and only if there is a closed disc C that contains pi and pj on its boundary and does not contain any other point of P.

Theorem 9.6 readily implies the following characterization of Delaunay triangulations.

Theorem 9.7 Let P be a set of points in the plane, and let T be a triangulation of P. Then T is a Delaunay triangulation of P if and only if the circumcircle of any triangle of T does not contain a point of P in its interior.

Since we argued before that a triangulation is good for the purpose of height interpolation

if its angle-vector is as large as possible, our next step should be to look at the angle-vector of Delaunay triangulations. We do this by a slight detour through legal triangulations.

Theorem 9.8 Let P be a set of points in the plane. A triangulation T of P is legal if and only if T is a Delaunay triangulation of P.

Proof. It follows immediately from the definitions that any Delaunay triangulation is legal.

We shall prove that any legal triangulation is a Delaunay triangulation by contradiction. So assume T is a legal triangulation of P that is not a Delaunay triangulation. By Theorem 9.6, this means that there is a triangle pipjpk such that the circumcircle C(pipjpk) contains a point pl e P in its interior. Let e := pipj be the edge of pipjpl such that the triangle pipjpl does not intersect pipjpk. Of all such pairs (pipjpk, pl) in T, choose the one that maximizes the angle Åpiplpj. Now look at the triangle pipjpm adjacent to

pipjpk along e. Since T is legal, e is legal. By Lemma 9.4 this implies that pm does not lie in the interior of C(pipjpk). The circumcircle C(pipjpm) of pipjpm contains the part of C(pipjpk) that is separated frompipjpk by e. Consequently, pl e C(pipjpm). Assume that pjpm is the edge of pipjpm such that pjpmpl does not intersect pipjpm. But now Åpjplpm > Åpiplpj by Thales's Theorem, contradicting the definition of the pair (pipjpk, pl). EO

Since any angle-optimal triangulation must be legal, Theorem 9.8 implies that any angle-optimal triangulation of P is a Delaunay triangulation of P. When P is in general position, there is only one legal triangulation, which is then the only angle-optimal triangulation, namely the unique Delaunay triangulation that coincides with the Delaunay graph. When P is not in general position, then any triangulation of the Delaunay graph is legal. Not all these Delaunay triangulations need to be angle-optimal. However, their angle-vectors do not differ too much. Moreover, using Thales's Theorem one can show that the minimum angle in any triangulation of a set of co-circular points is the same,

that is, the minimum angle is independent of the triangulation. This implies that any triangulation turning the Delaunay graph into a Delaunay triangulation has the same minimum angle. The following theorem summarizes this.

Theorem 9.9 Let P be a set of points in the plane. Any angle-optimal trian-gulation of P is a Delaunay triangulation of P. Furthermore, any Delaunay triangulation of P maximizes the minimum angle over all triangulations of P.

9.3 Computing the Delaunay Triangulation

We have seen that for our purpose—approximating a terrain by constructing a polyhedral terrain from a set P of sample points—a Delaunay triangulation of P is a suitable triangulation. This is because the Delaunay triangulation maximizes the minimum angle. So how do we compute such a Delaunay triangulation?

We already know from Chapter 7 how to compute the Voronoi diagram of P. From Vor(P) we can easily obtain the Delaunay graph DG(P), and by triangulating the faces with more than three vertices we can obtain a Delaunay

triangulation. In this section we describe a different approach: we will compute a Delaunay triangulation directly, using the randomized incremental approach we have so successfully applied to the linear programming problem in Chapter 4 and to the point location problem in Chapter 6.

In Chapter 6 we found it convenient to start with a large rectangle containing the scene, to avoid problems caused by unbounded trapezoids. In the same spirit we now start with a large triangle that contains the set P. We will add two extra points p-1 and p-2 that, together with the highest point p0 of P, form a triangle containing all the points. This means we are now computing a Delaunay triangulation of P ^J{p-1, p-2 } instead of the Delaunay triangulation of P.

Later we want to obtain the Delaunay triangulation of P by discarding p-1 and p-2, together with all incident edges. For this to work we have to choose p-1 and p-2 far enough away, so that they don’t destroy any triangles in the Delaunay triangulation of P. In particular, we must ensure

they do not lie in any circle defined by three points in P. We postpone the details of this to a later stage; first we have a look at the algorithm.

The algorithm is randomized incremental, so it adds the points in random order and it maintains a Delaunay triangulation of the current point set. Consider the addition of a point pr. We first find the triangle of the current triangulation that contains pr—how this is done will be explained later—and we add edges from pr to the vertices of this triangle. If pr happens to fall on an edge e of the triangulation, we have to add edges from pr to the opposite vertices in the triangles sharing e. Figure 9.7 illustrates these two cases. We now have pr lies in the interior of a triangle pr falls on an edge remedy this, we call a procedure LegalizeEdge with each potentially illegal edge. This procedure replaces illegal edges by legal ones through edge flips. Before we come to the details of this, we give a precise description of the main algorithm. It will be convenient for the analysis to let P be a set of n +1 points.

Algorithm DELAUNAYTRIANGULATION(P)

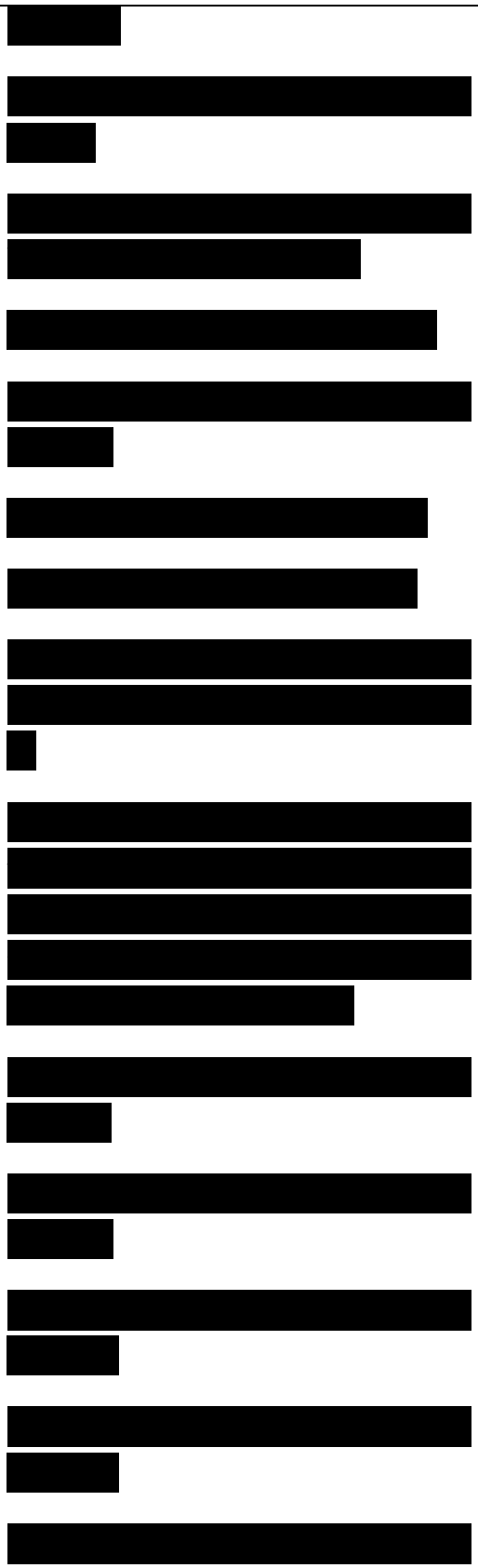
Input. A set P of n + 1 points in the plane.

Output. A Delaunay triangulation of P.

1. Let p0 be the lexicographically highest point of P, that is, the rightmost among the points with largest y-coordinate.
2. Let p_1 and p_2 be two points in R2 sufficiently far away and such that P is contained in the triangle p0p_1 p_2.
3. Initialize T as the triangulation consisting of the single triangle p0p_1 p_2.

4. Compute a random permutation p 1, p2pn of P \ {p0}.
5. for r^- 1 to n
6. do (* Insert pr into T: *)

7. Find a triangle pipjpk e T containing pr.

8. if pr lies in the interior of the triangle pipjpk 9. then Add edges from pr to the three vertices of pipjpk, thereby splitting pipjpk into three triangles. 10. LegaliZeEDge(p , pip], T) 11. legalizeEdge(pr, p]pk, T) 12. legalizeEdge(pr, pkpi, T) 13. else (* pr lies on an edge of pipjpk, say the edge pip] *) 14. Add edges from pr to pk and to the third vertex pi of the other triangle that is incident to pip], thereby splitting the two triangles incident to pip] into four triangles. 15. LegalizeEdge(pr, pipi, T) 16. LegalizeEdge(pr, pip], T) 17. LegalizeEdge(pr, pjpk, T) 18. LegalizeEdge(pr, pkpi, T) 19. Discard p_1 and p_2 with all their incident edges		

from T.

20. return T

Next we discuss the details of turning the triangulation we get after line 9 (or line 14) into a Delaunay triangulation. We know from Theorem 9.8 that a triangulation is a Delaunay triangulation if all its edges are legal. In the spirit of algorithm LegalTriangulation, we therefore flip illegal edges until the triangulation is legal again.

The question that remains is which edges may become illegal due to the insertion of pr. Observe that an edge pTpj that was legal before can only become illegal if one of the triangles incident to it has changed.

So only the edges of the new triangles need to be checked. This is done using the subroutine LegalizeEdge, which tests and possibly flips an edge. If LegalizeEdge flips an edge, other edges may become illegal. Therefore LegalizeEdge calls itself recursively with such potentially illegal edges.

LegalizeEdge(pr, pĩpĩ, T)
1. (* The point being inserted is pr, and pipj is the edge of T that may need to be flipped. *)

2. if pp is illegal
3. then Let pipjpk be the triangle adjacent to prpipj along ppj.
4. (* Flip pipj: *) Replace pTpj with prpk.
5. LegalizeEdge(pr, pipk, T)
6. LegalizeEdge(pr, pkpi, T)

The test in line 2 whether an edge is illegal can normally be done by applying Lemma 9.4. There are some complications because of the presence of the special points p-1 and p-2. We shall come back to this later; first we prove that the algorithm is correct.

Figure 9.8
All edges created are incident to pr

To ensure the correctness of the algorithm, we need to prove that no illegal edges remain

after all calls to LegalizeEdge have been processed. From the code of LegalizeEdge it is clear that every new edge created due to the insertion of pr is incident to pr. Figure 9.8 illustrates this; the triangles that are destroyed and the new triangles are shown in grey. The crucial observation (proved below) is that every new edge must be legal, so there is no need to test them. Together with the earlier observation that an edge can only become illegal if one of its incident triangles changes, this proves that the algorithm tests any edge that may become illegal. Hence, the algorithm is correct. Note that, as in Algorithm LegalTriangulation, the algorithm cannot get into an infinite loop, because every flip makes the angle-vector of the triangulation larger.
{p_2, p_1, p0,..., pr}.

Proof. Consider first the edges prpi, prp], prpk (and perhaps prpi) created by splitting pipipk (and maybe pipipl). Since pipipk is a triangle in the Delaunay triangulation before the addition of pr, the circumcircle C of ppp contains

no point pt with t < r in its interior. By shrinking C we can find a circle C through pi and pr contained in C. Because C' c C we know that C' is empty. This implies that prpi is an edge of the Delaunay graph after the addition of pr. The same holds for prp] and prpk (and for prpi, if it exists). Now consider an edge flipped by LegalizeEdge. Such an edge flip always replaces an edge pip] of a triangle pipipl by an edge prpi incident to pr. Since pipipi was a Delaunay triangle before the addition of pr and because its circumcircle C contains pr—otherwise pip] would not be illegal—we can shrink the circumcircle to obtain an empty circle C' with only pr and pi on its boundary. Hence, prpl is an edge of the Delaunay graph after the addition. EQ We have proved the correctness of the algorithm. What remains is to describe how to implement two important steps: how to find the triangle containing the point pr in line 7 of DelaunayTriangulation, and		

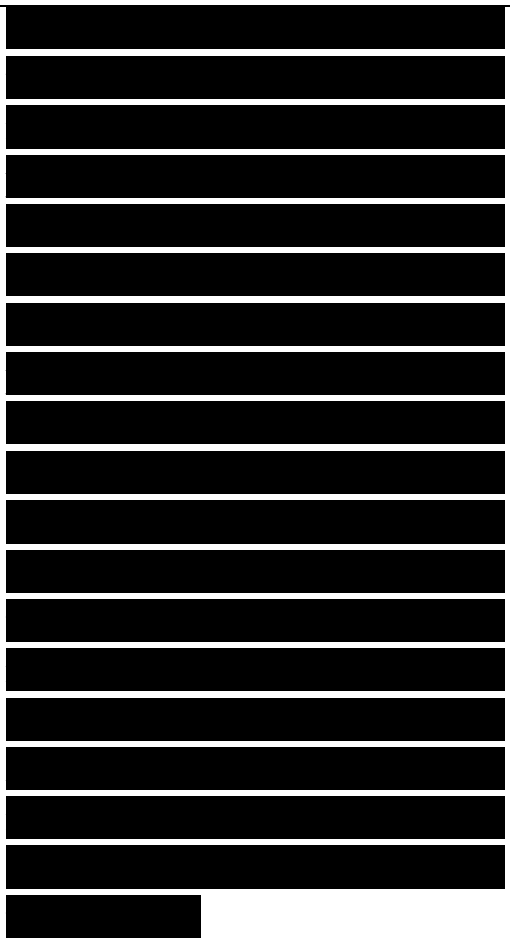
how to deal correctly with the points p_1 and p_2 in the test in line 2 in LegalizeEdge. We start with the former issue.

To find the triangle containing pr we use an approach quite similar to what we did in Chapter 6: while we build the Delaunay triangulation, we also build a point location structure D, which is a directed acyclic graph. The leaves of D correspond to the triangles of the current triangulation T, and we maintain cross-pointers between those leaves and the triangulation.

The internal nodes of D correspond to triangles that were in the triangulation at some earlier stage, but have already been destroyed. The point location structure is built as follows. In line 3 we initialize D as a DAG with a single leaf node, which corresponds to the triangle p0p_1 p_2.

Now suppose that at some point we split a triangle pipipk of the current triangulation into three (or two) new triangles. The corresponding change in D is to add three (or two) new

leaves to D, and to make the leaf for pipipk into an internal node with outgoing pointers to those three (or two) leaves. Similarly, when we replace two triangles pkpipi and pipipl by triangles pkpipl and pkplp] by an edge flip, we create leaves for the two new triangles, and the nodes of pkpipi and ppp get pointers to the two new leaves. Figure 9.9 shows an example of the changes in D caused by the addition of a point. Observe that when we make a leaf into an internal node, it gets at most three outgoing pointers.



Using D we can locate the next point pr to be added in the current triangulation. This is done as follows. We start at the root of D, which corresponds to the initial triangle p0p_1 p_2. We check the three children of the root to see in which triangle pr lies, and we descend to the corresponding child. We then

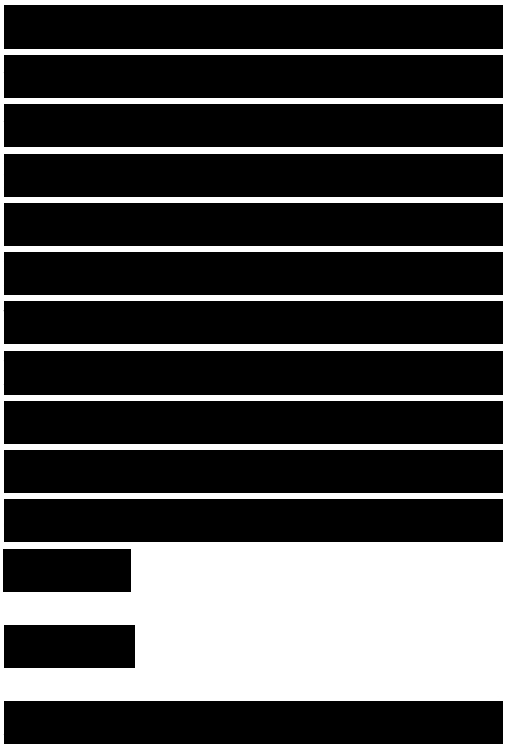


Figure 9.9
The effect of inserting point pr into triangle À1 on the data

structure D (the part of D that does not change is omitted in the figure)

check the children of this node, descend to a child whose triangle contains pr, and so on, until we reach a leaf of D. This leaf corresponds to a triangle in the current triangulation that contains pr. Since the out-degree of any node is at most three, this takes linear time in the number of nodes on the search path, or, in other words, in the number of triangles stored in D that contain pr.

There is only one detail left, namely how to choose p_1 and p_2, and how to implement the test of whether an edge is legal. On the one hand, we have to choose p_1 and p_2 to be far away, because we don't want their presence to influence the Delaunay triangulation of P. One the other hand, we don't want to introduce the huge coordinates needed for that. So what we do is to treat these points symbolically: we do not actually assign coordinates to them, but instead modify the tests for point location and for illegal edges such that they work as if we had chosen the points to be very far away.

In the following, we will say that p = (xp,yp) is higher than q = (xq,yq) if yp > yq or yp = yq and xq > xp, and use the (lexicographic) ordering on P induced by this relation.

Let l_1 be a horizontal line lying below the entire set P, and let t_2 be a horizontal line lying above P. Conceptually, we choose p_1 to lie on the line l_1 sufficiently far to the right that p_1 lies outside every circle defined by three non-collinear points of P, and such that the clockwise ordering of the points of P around p_1 is identical to their (lexicographic) ordering.

Next, we choose p_2 to lie on the line t_2 sufficiently far to the left that p_2 lies outside every circle defined by three non-collinear points of P u{p_1}, and such that the counterclockwise ordering of the points of P u{p_1} around p_2 is identical to their (lexicographic) ordering.

The Delaunay triangulation of

P u {p_1: p_2} consists of the Delaunay triangulation of P, edges connecting p_1 to every point on the right convex hull of P, edges connecting p_2 to every point on the left convex hull of P, and the one edge p_1 p_2. The lowest point of P and the highest point p0 of P are connected to both p_1 and p_2.

During the point location step, we need to determine the position of a point pj with respect to the oriented line from pi to pk. By our choice of p_1 and p_2, the following conditions are equivalent:

- pj lies to the left of the line from pi to p_1;
- pj lies to the left of the line from p_2 to pf;
- pj is lexicographically larger than pi.

It remains to explain how to treat p_1 and p_2 when we check whether an edge is illegal. Let pp be the edge to be tested, and let pk and pl be the other vertices of the triangles incident to pTpj (if they exist).

- pTpj is an edge of the triangle p0 p_1 p_2. These edges are always legal.

■ The indices i, j, k, l are all non-negative. This is the normal case; none of the points involved in the test is treated symbolically. Hence, pTpj is illegal if and only if pl lies inside the circle defined by pi, pj, and pk.

■ All other cases. In this case, pipj is legal if and only if min(k, l) < min(i, j).

Only the last case requires further justification. Since the situation where pipj is p_1 p_2 is handled in the first case, at most one of the indices i and j is negative. On the other hand, either pk or pl is the point pr that we have just inserted, and so at most one of the indices k and l is negative.

If only one of the four indices is negative, then this point lies outside the circle defined by the other three points, and the method is correct.

Otherwise, both min(i,]) and min(k, l) are negative, and the fact that p_2 lies outside any circle defined by three points in Pu {p_]_} implies that the method is correct.

9.4 The Analysis

We first look at the structural change generated by the algorithm. This is the number of triangles created and deleted during the course of the algorithm. Before we start the analysis, we introduce some notation: Pr := {p1,..., pr} and DSr := Dg({p_2,p_1,p0}UPr).

Lemma 9.11 The expected number of triangles created by algorithm DELAU-NAYTRIANGULATION is at most 9n + 1.

Proof. In the beginning, we create the single triangle p0p_1 p_2. In iteration r of the algorithm, when we insert pr, we first split one or two triangles, creating three or four new triangles. This splitting creates the same number of edges in D3r, namely prpi, prp], prpk (and maybe prpl). Furthermore, for every edge that we flip in procedure LegalizeEdge, we create two new triangles. Again, the flipping creates an edge of D3r incident to pr. To summarize: if after the insertion of pr there are k edges of Dgr incident to pr, then we have created at

most 2(k _ 3) + 3 = 2k _ 3 new triangles. The number k is the degree of pr in D3r; we denote this degree by deg(pr, DGr). degree of pr, over all possible permutations of the set P? As in Chapter 4 and 6 we use backwards analysis to bound this value. So, for the moment, we fix the set Pr. We want to bound the expected degree of the point pr, which is a random element of the set Pr.

By Theorem 7.3, the Delaunay graph Dgr has at most 3(r + 3) _ 6 edges. Three of these are the edges of p0p_1 p_2, and therefore the total degree of the vertices in Pr is less than 2[3(r + 3) _ 9]= 6r. This means that the expected degree of a random point of Pr is at most 6. Summarizing the above, we can bound the number of triangles created in step r as follows.

E [number of triangles created in step r] < E[2deg(pr, DGr) _ 3]
= 2E[deg(pr,DSr)] _ 3
< 2 • 6 _ 3 = 9

The total number of created triangles is one for the triangle p0p_1 p_2 that we start with, plus the number of triangles

created in each of the insertion steps. Using linearity of expectation, we get that the expected total number of created triangles is bounded by 1 + 9n. EÕ

We now state the main result.

Theorem 9.12 The Delaunay triangulation of a set P of n points in the plane DELAUNAY TRIANGULATIONS can be computed in O(nlogn) expected time, using O(n) expected storage.

Proof. The correctness of the algorithm follows from the discussion above. As for the storage requirement, we note that only the search structure D could use more than linear storage. However, every node of D corresponds to a triangle created by the algorithm, and by the previous lemma the expected number of these is O(n).

To bound the expected running time we first ignore the time spent in the point location step (line 7). Now the time spent by the algorithm is proportional to the number of created triangles.

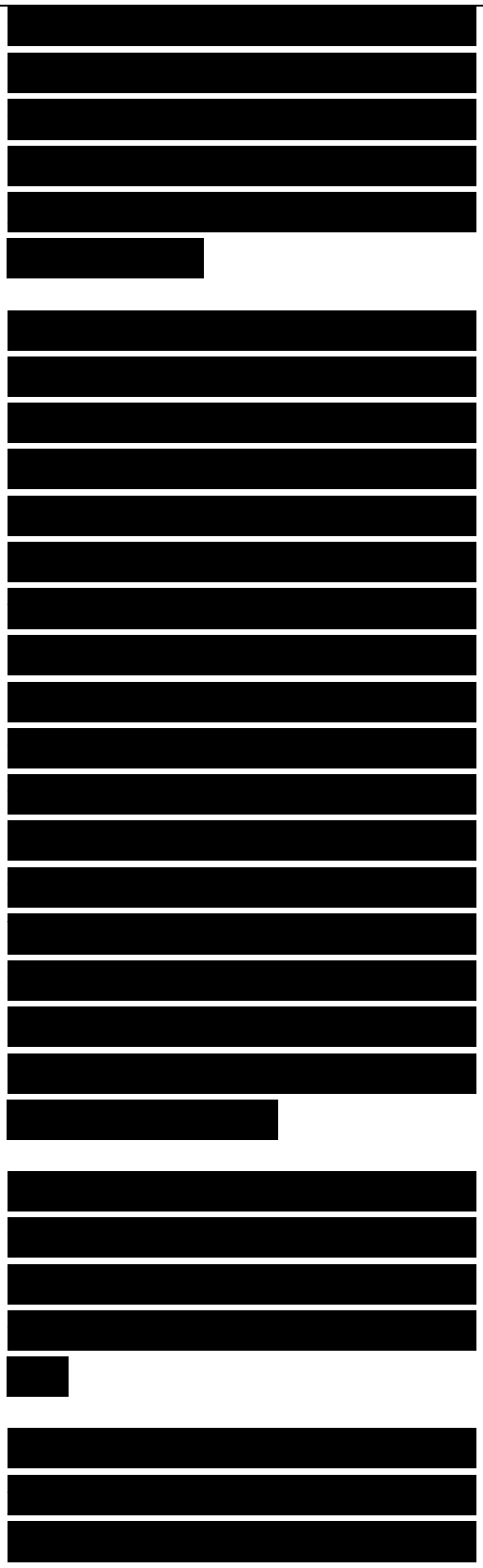
From the previous lemma we can therefore conclude that the expected running time, not counting the time for point location, is O(n).

It remains to account for the point location steps. The time to locate the point pr in the current triangulation is linear in the number of nodes of D that we visit. Any visited node corresponds to a triangle that was created at some earlier stage and that contains pr.

If we count the triangle of the current triangulation separately, then the time for locating pr is O(1) plus linear time in the number of triangles that were present at some earlier stage, but have been destroyed, and contain pr.
A triangle pipipk can be destroyed from the triangulation for one of two reasons:

- A new point pl has been inserted inside (or on the boundary of) pipipk, and ppp was split into three (or two) subtriangles.

- An edge flip has replaced pipipk and an adjacent triangle pipipl by the pair pkpipl and pkp]pl.



In the first case, the triangle pip]pk was a Delaunay triangle before pl was inserted.

In the second case, either pipipk was a Delaunay triangle and pl was inserted, or pipipl was a Delaunay triangle and pk was inserted. If pipipl was the Delaunay triangle, then the fact that the edge pip] was flipped means that both pk and pr lie inside the circumcircle of pipipl.

In all cases we can charge the fact that triangle pipipk was visited to a Delaunay triangle A that has been destroyed in the same stage as pipipk, and such that the circumcircle of A contains pr. Denote the subset of points in P that lie in the circumcircle of a given triangle A by K(A).

In the argument above the visit to a triangle during the location of pr is charged to a triangle A with pr e K(A). It is easy to see that a triangle A can be charged at most once for every one of the points in K(A). Therefore the total time for the point location steps is

where the summation is over all Delaunay triangles A created by the algorithm. We shall prove later that the expected value of this sum is O(nlogn). This proves the theorem. ED

It remains to bound the expected size of the sets K(A). If A is a triangle of the Delaunay triangulation D3r, then what would we expect card(K(A)) to be?

For r = 1 we would expect it to be roughly n, and for r = n we know that it is zero. What happens in between? The nice thing about randomization is that it "interpolates" between those two extremes. The right intuition would be that, since Pr is a random sample, the number of points lying inside the circumcircle of a triangle À e DGr is about O(n/r). But be warned: this is not really true for all triangles in Dgr. Nevertheless, the sum in expression (9.1) behaves as if it were true.

In the remainder of this section we will give a quick proof of this fact for the case of a point

set in general position. The result is true for the general case as well, but to see that we have to work a little bit harder, so we postpone that to the next section, where we treat the problem in more generality.

Lemma 9.13 If P is a point set in general position, then

where the summation is over all Delaunay triangles À created by the algorithm.

Proof. Since P is in general position, every subset Pr is in general position. This implies that the triangulation after adding the point pr is the unique triangulation DGr. We denote the set of triangles of Dgr by Tr. Now the set of Delaunay triangles created in stage r equals Tr \ Tr_1 by definition. Hence, we can rewrite the sum we want to bound as

For a point q, let k(Pr, q) denote the number of triangles À e Tr such that q e K(À), and let k(Pr, q, pr) be the number of triangles À e Tr such that not

only q e K(À) but for which we also have that pr is incident to À. Recall that any Delaunay triangle created in stage r is incident to pr, so we have

For the moment, we fix Pr. In other words, we consider all expectations to be over the set of permutations of the set P where Pr is equal to a fixed set P*. The value of k(Pr, q, pr) then depends only on the choice of pr. Since a triangle À e Tr is incident to a random point p e P* with probability at most 3/r, we get

If we sum this over all q e P \ Pr and use (9.2), we get

Every q e P \ Pr is equally likely to appear as pr+1, and so we have

We can substitute this into (9.3), and get

What is k(Pr, pr+1)? It is the number of triangles A of Tr that have pr+1 e K(A). By the criterion from Theorem 9.6 (i), these triangles are exactly the triangles of Tr that will be destroyed by the insertion of pr+1. Hence, we can rewrite

the previous expression as

Theorem 9.1 shows that the number of triangles in Tm is precisely 2(m + 3) _ 2 _ 3 = 2m + 1.
Therefore, the number of triangles destroyed by the insertion of point pr+1 is exactly two less than the number of triangles created by the insertion of pr+1, and we can rewrite the sum as
Until now we considered Pr to be fixed. At this point, we can simply take the average over all choices of Pr c P on both sides of the inequality above, and find that it also holds if we consider the expectation to be over all possible permutations of the set P.

We already know that the number of triangles created by the insertion of pr+1 is identical to the number of edges incident to pr+1 in Tr+1, and that the expected number of these edges is at most 6. We conclude that
Summing over r proves the lemma. 0

9.5* A Framework for Randomized Algorithms
Up to now we have seen three randomized incremental algorithms in this book: one for linear programming in Chapter

4, one for computing a trapezoidal map in Chapter 6, and one for computing a Delaunay triangulation in this chapter. (We will see one more in Chapter 11.) These algorithms, and most other randomized incremental algorithms in the computational geometry literature, all work according to the following principle.

Suppose the problem is to compute some geometric structure T(X), defined by a set X of geometric objects. (For instance, a Delaunay triangulation defined by a set of points in the plane.) A randomized incremental algorithm does this by adding the objects in X in random order, meanwhile maintaining the structure T. To add the next object, the algorithm first finds out where the current structure has to be changed because there is a conflict with the object—the location step—and then it updates the structure locally—the update step. Because all randomized incremental algorithms are so much alike, their analyses are quite similar as well. To avoid having to prove the same bounds over and over again for different problems, an

axiomatic framework has been developed that captures the essence of randomized incremental algorithms. This framework—called a configuration space—can be used to prove ready-to-use bounds for the expected running time of many randomized incremental algorithms. (Unfortunately, the term "configuration space" is also used in motion planning, where it means something completely different—see Chapter 13.) In this section we describe this framework, and we give a theorem that can be used to analyze any randomized incremental algorithm that fits into the framework. For instance, the theorem can immediately be applied to prove Lemma 9.13, this time without assuming that P has to be in general position.

A configuration space is defined to be a four-tuple (X, n, D, K). Here X is the input to the problem, which is a finite set of (geometric) objects; we denote the cardinality of X by n. The set n is a set whose elements are called configurations. Finally, D and K both assign to every

configuration À e n a subset of X, denoted D(À) and K(À) repectively. Elements of the set D(À) are said to define the configuration À, and the elements of the set K(À) are said to be in conflict with, or to kill, À. The number of elements of K(À) is called the conflict size of the configuration À. We require that (X, n, D, K) satisfies the following conditions.

■ The number d := max{card(D(À)) I À e n} is a constant. We call this number the maximum degree of the configuration space. Moreover, the number of configurations sharing the same defining set should be bounded by a constant.

■ We have D(À) n K(À) = 0 for all configurations À e n.

A configuration À is called active over a subset S c X if D(À) is contained in S and K(À) is disjoint from S. We denote the set of configurations active over S by T(S), so we have

T(S) := {À e n : D(À) c S and K(À) n S = (0}.

The active configurations form the structure we want to compute. More precisely, the

goal is to compute T(X).

Before we continue our discussion of this abstract framework, let's see how the geometric structures we have met so far fit in.

Half-plane intersection. In this case the input set X is a set of half-planes in the plane. We want to define n, D, and K in such a way that T(X) is what we want to compute, namely the intersection of the half-planes in X.

We can achieve this as follows. The set n of configurations consists of all the intersection points of the lines bounding the half-planes in X. The defining set D(À) of a configuration À e n consists of the two lines defining the intersection, and the killing set K(À) consists of all half-planes that do not contain the intersection point. Hence, for any subset S c X, and in particular for X itself, T(S) is the set of vertices of the common intersection of the half-planes in S.

Trapezoidal maps.

Here the input set X is a set of segments in the plane. The set n of configurations contains all trapezoids appearing in the trapezoidal map of any S c X. The defining set D(A) of a configuration A is the set of segments that are necessary to define A. The killing set K(A) of a trapezoid A is the set of segments that intersect A. With these definitions, T(S) is exactly the set of trapezoids of the trapezoidal map of S.

Delaunay Triangulation. The input set X is a set of points in general position in the plane. The set n of configurations consists of triangles formed by three (non-collinear) points in X. The defining set D(A) consists of the points that form the vertices of A, and the killing set K(A) is the set of points lying inside the circumcircle of A. By Theorem 9.6, T(S) is exactly the set of triangles of the unique Delaunay triangulation of S.

As stated earlier, the goal is to compute the structure T(X). Randomized incremental algorithms do this by computing a random

permutation x1,x2,...,xn of the objects in X and then adding the objects in this order, meanwhile maintaining T(Xr), where Xr := {x1; x2,... ,xr}. The fundamental property of configuration spaces that makes this possible is that we can decide whether or not a configuration A appears in T(Xr) by looking at it locally—we only need to look for the defining and killing objects of A. In particular, T(Xr) does not depend on the order in which the objects in Xr were added.

For instance, a triangle A is in the Delaunay triangulation of S if and only if the vertices of A are in S, and no point of S lies in the circumcircle of A.

The first thing we usually did when we analyzed a randomized incremental algorithm was to prove a bound on the expected structural change—see for instance Lemma 9.11. The next theorem does the same, but now in the abstract configuration-space framework.

Theorem 9.14 Let (X, n, D, K) be a configuration space, and let T and Xr be defined as above. Then the expected

number of configurations in T(Xr) \ T(Xr_1) is at most

where d is the maximum degree of the configuration space.

Proof. As in previous occasions where we wanted to bound the structural change, we use backwards analysis: instead of trying to argue about the number of configurations that appear due to the addition of xr into Xr_ 1, we shall argue about the number of configurations that disappear when we remove xr from Xr. To this end we temporarily let Xr be some fixed subset X* c X of cardinality r. We now want to bound the expected number of configurations A e T(Xr) that disappear when we remove a random object xr from Xr. By definition of T, such a configuration A must have xr e D(A). Since there are at most d • card(T(Xr)) pairs (x, A) with A e T(Xr) and x e D(A), we have

Hence, the expected number of configurations disappearing due to the removal of a random object from Xr is at most - card(T(Xr)). In this argument, the set Xr was a fixed subset X* c X of cardinality r. To

obtain the general bound, we have to average over all possible subsets of size r, which gives a bound of d E [card(T (Xr))]. □

This theorem gives a generic bound for the expected size of the structural change during a randomized incremental algorithm. But what about the cost of the location steps? In many cases we will need a bound of the same form as in this chapter, namely we need to bound

where the summation is over all configurations À that are created by the algorithm, that is, all configurations that appear in one of the T(Xr). This bound is given in the following theorem.

Theorem 9.15 Let (X, n, D, K) be a configuration space, and let T and Xr be defined as above. Then the expected value of

where the summation is over all configurations À appearing in at least one T(Xr) with $1 < r < n$, is at most

where d is the maximum degree of the configuration space.

Proof. We can follow the proof of Lemma 9.13 quite closely.

We first rewrite the sum as

Next, let k(Xr,y) denote the number of configurations À e T(Xr) such that y e K(À), and let k(Xr, y, xr) be the number of configurations À e T(Xr) such that not only y e K(À) but for which we also have xr e D(À). Any new configuration appearing due to the addition of xr must have xr e D(À). This implies that

We now fix the set Xr. The expected value of k(Xr,y,xr) then depends only on the choice of xr e Xr. Since the probability that y e D(À) for a configuration

À e T(Xr) is at most d/r, we have

If we sum this over all y e X \ Xr and use (9.4), we get

On the other hand, every y e X \ Xr is equally likely to appear as xr+1, so E[k(Xr,

Substituting this into (9.5) gives

Now observe that k(Xr, xr+1)

is the number of configurations A of T(Xr) that will be destroyed in the next stage, when xr+1 is inserted. This means we can rewrite the last expression as

Unlike in the proof of Lemma 9.13, however, we cannot simply bound the number of configurations destroyed in stage r + 1 by the number of configurations created at that stage, because that need not be true in a general configuration space. Hence, we proceed somewhat differently.

First we observe that we can take the average over all choices of Xr on both sides of (9.6) and find that it also holds if the expectation is over all permutations of X. Next, we sum over all r, and rewrite the sum as follows:

where the summation on the right hand side is over all configurations A that are created and later destroyed by the algorithm, and where j(A) denotes the stage when configuration A is destroyed. Let i(A) denote the stage when the configuration A is created.

Since i(A) < j(A) _ 1, we have

If we substitute this into (9.7), we see that

The right hand side of this expression is at most

(the difference being only those configurations that are created but never de-stroyed) and so we have

By Theorem 9.14, we get the bound we wanted to prove:

This finishes the analysis in the abstract setting. As an example, we will show how to apply the results to our randomized incremental algorithm for computing the Delaunay triangulation. In particular, we will prove that

where the summation is over all triangles À created by the algorithm, and where K(À) is the set of points in the circumcircle of the triangle.

Unfortunately, it seems impossible to properly define a configuration space whose configurations are triangles when the points are not in general position. Therefore we

shall choose the configurations slightly differently.

Let P be a set of points in the plane, not necessarily in general position. Let Q := {p0, p—1, p_2} denote the set of three points we used to start the construction. Recall that Po is the lexicographically largest point from P, while points p_1 and p-2 were chosen such that they do not destroy any Delaunay edges between points in P. We set X := P\{p0}. Every triple À = (pi, pj, pk) of points in X u Q that do not lie on a line defines a configuration with D(À) := {pi,pj,pk}nX and K(À) is the set of points of X that lie either in the interior of the circumcircle of the triangle pi p j pk or on the circular arc on the circumcircle from pi to pk containing pj.

We call such a configuration À a Delaunay corner of X, because À is active over S c X if and only if pi, pj, and pk are consecutive points on the boundary of one face of the Delaunay graph DS(Q u S). Note that any set of three non-collinear points defines three different configurations.

The important observation is that whenever DelaunayTriangulation creates a new triangle, this triangle is of the form piprpj, where pr is the point inserted in this stage, and prp and prpj are edges of the Delaunay graph DS(Q uPr)—see Lemma 9.10. It follows that when the triangle piprpj is created, the triple (pj,pr,pi) is a Delaunay corner of DS(Q uPr) and, hence, it is an active configuration over the set Pr. The set K(À) defined for this configuration contains all points contained in the circumcircle of the triangle piprpj. We can therefore bound the original sum by

where the sum is over all Delaunay corners À that appear in some intermediate Delaunay graph DS (Q u Pr).

Now Theorem 9.15 applies. How many Delaunay corners are there in the TRIANGULATIONS Delaunay graph of S u Q? The worst case is when the Delaunay graph is a triangulation. If S contains r points, then the triangulation has 2(r + 3) _ 5 triangles, and

therefore 6(r + 3) _ 15 = 6r + 3 Delaunay corners. It follows from Theorem 9.15 that
This finally completes the proof of Theorem 9.12.

9.6 Notes and Comments

The problem of triangulating a set of points is a topic in computational geometry that is well known outside this field. Triangulations of point sets in two and more dimensions are of paramount importance in numerical analysis, for instance for finite element methods, but also in computer graphics. In this chapter we looked at the case of triangulations that only use the given points as vertices. If additional points—so-called Steiner points—are allowed, the problem is also known as meshing and is treated in more detail in Chapter 14.

Lawson [244] proved that any two triangulations of a planar point set can be transformed into each other by flipping edges. He later suggested finding a good triangulation by iteratively flipping edges,

where such an edge-flip improves some cost function of the triangulation [245].

It had been observed for some time that triangulations that lead to good interpolations avoid long and skinny triangles [38]. The result that there is—if we ignore degenerate cases—only one locally optimal triangulation with respect to the angle-vector, namely the Delaunay triangulation, is due to Sibson [360].

Looking only at the angle-vector completely ignores the height of the data points, and is therefore also called the data-independent approach. A good motivation for this approach is given by Rippa [328], who proves that the Delaunay triangulation is the triangulation that minimizes the roughness of the resulting terrain, no matter what the actual height data is. Here, roughness is defined as the integral of the square of the L2-norm of the gradient of the terrain. More recent research tries to find improved triangulations by taking the height information into account. This data-dependent approach was first proposed by Dyn et al. [154], who suggest

different cost criteria for triangulations, which depend on the height of the data points.

Interestingly, they compute their improved triangulations by starting with the Delaunay triangulation and iteratively flipping edges. The same approach is taken by Quak and Schumaker [325], who consider piecewise cubic interpolation, and Brown [76]. Quak and Schumaker observe that their triangulations are small improvements compared to the Delaunay triangulation when they try to approximate smooth surfaces, but that they can be drastically different for non-smooth surfaces.

More references relevant to Delaunay triangulations as the dual of Voronoi diagrams can be found in Chapter 7.

The randomized incremental algorithm we have given here is due to Guibas Section 9.7 et al. [196], but our analysis of £Acard(K(A)) is from Mulmuley's book [290]. The argument that extends the analysis to the case of points in

degenerate position is new. Alternative randomized algorithms were given by Boissonnat et al. [69, 71], and by Clarkson and Shor [133].

Various geometric graphs defined on a set of points P have been found to be subgraphs of the Delaunay triangulation of P. The most important one is probably the Euclidean minimum spanning tree (EMST) of the set of points [349]; others are the Gabriel graph [186] and the relative neighborhood graph [374].

We treat these geometric graphs in the exercises.

Another important triangulation is the minimum weight triangulation, that is, a triangulation whose weight is minimal (where the weight of a triangulation is the sum of the lengths of all edges of the triangulation) [12, 42, 146, 147].
Determining a minimum weight triangulation among all triangulations of a given point set was recently shown to be NP-complete [291].

9.7 Exercises

9.1 In this exercise we look

at the number of different triangulations that a set of n points in the plane may allow.

a. Prove that no set of n points can be triangulated in more than 2(n) ways.

b. Prove that there are sets of n points that can be triangulated in at least 2-_2v'- different ways.

9.2 The degree of a point in a triangulation is the number of edges incident to it. Give an example of a set of n points in the plane such that, no matter how the set is triangulated, there is always a point whose degree is n - 1.

9.3 Prove that any two triangulations of a planar point set can be transformed into each other by edge flips. Hint: Show first that any two triangulations of a convex polygon can be transformed into each other by edge flips.

9.4 Prove that the smallest angle of any triangulation of a convex polygon whose vertices lie on a circle is the same. This implies that any completion of the Delaunay triangulation of a

set of points maximizes the minimum angle.

9.5 a. Given four points p, q, r, s in the plane, prove that point s lies in the interior of the circle through p, q, and r if and only if the following condition holds. Assume that p, q, r form the vertices of a triangle in clockwise order.

b. The determinant test of part a. can be used to test if an edge in a triangulation is legal. Can you come up with an alternative way to implement this test? Discuss the advantages and/or disadvantages of your method compared to the determinant test.

9.6 We have described algorithm DELAUNAYTRIANGULATION by calling a recursive procedure LEGALIZEEDGE. Give an iterative version of this procedure, and discuss the advantages and/or disadvantages of your procedure over the recursive one.

9.7 Prove that all edges of

DS(Pr) that are not in DS(Pr_1) are incident to pr. In other words, the new edges of DS(Pr) form a star as in Figure

9.8. Give a direct proof, without referring to algorithm DelaunayTriangu- LATION.

9.8 Let P be a set of n points in general position, and let q e P be a point inside the convex hull of P. Let pi, p j, pk be the vertices of a triangle in the Delaunay triangulation of P that contains q. (Since q can lie on an edge of the Delaunay triangulation, there can be two such triangles.) Prove that qpi, qpj, and qpk are edges of the Delaunay triangulation of
P u{q}.

9.9 The algorithm given in this chapter is randomized, and it computes the Delaunay triangulation of a set of n points in O(nlog n) expected time. Show that the worst-case running time of the algorithm is Q(n2).

9.10 The algorithm given in this chapter uses two extra points p—1 and p_2 to start the construction of the Delaunay triangulation. These points should not lie in any circle defined by three input points, and so far away that they see the points of P in their lexicographic order. These

conditions were enforced by implementing operations involving these points in a special way—see page 204. Compute explicit coordinates for the extra points such that this special implementation is not needed. Is this a better approach?

9.11 A Euclidean minimum spanning tree (EMST) of a set P of points in the plane is a tree of minimum total edge length connecting all the points. EMST's are interesting in applications where we want to connect sites in a planar environment by communication lines (local area networks), roads, railroads, or the like.

a. Prove that the set of edges of a Delaunay triangulation of P contains an EMST for P.

b. Use this result to give an O(nlogn) algorithm to compute an EMST for P.

9.12 The traveling salesman problem (TSP) is to compute a shortest tour visiting all points in a given point set. The traveling salesman problem is NP-hard. Show how to find a tour whose length is at most two times the optimal length, using the EMST defined in the previous exercise.

9.13 The Gabriel graph of a set P of points in the plane is defined as follows: Section 9.7 Two points p and q are connected by an edge of the Gabriel graph if and EXERCISES only if the disc with diameter pq does not contain any other point of P.

a. Prove that Dg(P) contains the Gabriel graph of P.

b. Prove that p and q are adjacent in the Gabriel graph of P if and only if the Delaunay edge between p and q intersects its dual Voronoi edge.

c. Give an O(nlogn) time algorithm to compute the Gabriel graph of a set of n points.

9.14 The relative neighborhood graph of a set P of points in the plane is defined as follows: Two points p and q are connected by an edge of the relative neighborhood graph if and only if

a. Given two points p and q, let lune(p, q) be the moon-shaped region formed as the intersection of the two circles around p and q whose radius is d(p,q). Prove that p and q are connected in the relative neighborhood graph if and only if lune(p, q) does not contain any point of P in its interior.

b. Prove that Dg(P) contains the relative neighborhood graph of P.

c. Design an algorithm to compute the relative neighborhood graph of a given point set.

9.15 Prove the following relationship between the edge sets of an EMST, of the relative neighborhood graph (RNG), the Gabriel graph (GG), and the Delaunay graph (Dg) of a point set P.

EMST c RNG c GG c Dg.

(See the previous exercises for the definition of these graphs.)

9.16 A fc-clustering of a set P of n points in the plane is a partitioning of P into fc non-empty subsets P1,...,Pfc. Define the distance between any pair Pi, Pj of clusters to be the minimum distance between one point from Pi and one point from Pj, that is,

We want to find a fc-clustering (for given fc and P) that maximizes the minimum distance between clusters.

a. Suppose the mimimum distance between clusters is achieved by points p e Pi and q e Pj. Prove that pq is an edge of the Delaunay triangulation ofP.

b. Give an O(nlog n) time algorithm to compute a fc-clustering maximizing the minimum distance between clusters. Hint: Use a Union-Find data structure.

9.17 The weight of a

triangulation is the sum of the lengths of all edges of the triangulation. A minimum weight triangulation is a triangulation whose weight is minimal. Disprove the conjecture that the Delaunay triangulation is a minimum weight triangulation.

9.18* Give an example of a geometric configuration space (X, n, D, K) where T(Xr) \ T(Xr+1) can be arbitrarily large compared to T(Xr+1) \ T(Xr).

9.19* Apply configuration spaces to analyze the randomized incremental algorithm of Chapter 6.

The output of oil wells is a mixture of several different components, and the proportions of these components vary between different sources. This can sometimes be exploited: by mixing together the output of different wells, one can produce a mixture with proportions that are particularly favorable for the refining process.

Let's look at an example. For simplicity we assume that we are only interested in two of the components—call them A and B—of our product. Assume that we are given a mixture li with 10% of component A and 35% of component B, and another mixture with 16% of A and 20% of B. Assume further that what we really need is a mixture that contains 12% of A and 30% of B. Can we produce this mixture from the given ones? Yes, mixing |1 and |2 in the ratio 2 : 1 gives the desired product. However, it is impossible to make a mixture of |1 and |2 that contains 13% of A and 22% of B. But if we have a third mixture |3 containing 7% of A and 15% of B, then mixing |1, |2, and |3 in the ratio of 1: 3 : 1 will give the desired result.

What has all this to do with geometry? This becomes clear when we represent the mixtures |1, |2, and |3 by points in the plane, namely by p1 := (0.1,0.35), p2 := (0.16,0.2), and p3 := (0.07,0.15). Mixing |1 and in the ratio 2 : 1 gives the mixture represented by the point q := (2/3)p1 + (1/3)p2. This is the point on the segment prp2 such that dist(p2, q): dist(q, p1) = 2: 1, where dist(.,.) denotes the distance between two points. More generally, by mixing |1 and in varying ratios, we can produce the mixtures represented by any point on the line segment p1 p2. If we start with the three base mixtures |1, |2, and |3, we can produce any point in the triangle p1 p2p3. For instance, mixing |1, |2, and |3 in the ratio 1:3:1 gives the mixture represented by the point (1/5)p1 + (3/5)p2 + (1/5)p3 = (0.13,0.22).

What happens if we don't have three but n base mixtures, for some n > 3, represented by points p1;p2,...,pn? Suppose that we mix them in the ratio l1 : l2 : ■■■: ln. Let L := ĩ,tj=1 Ij and let Ǎi := li/L. Note that The mixture we get by

mixing the base mixtures in the given ratio is the one represented by

Such a linear combination of the points pi where the xi satisfy the conditions stated above—each xi is non-negative, and the sum of the xi is one—is called a convex combination. In Chapter 1 we defined the convex hull of a set of points as the smallest convex set containing the points or, more precisely, as the inter¬section of all convex sets containing the points. One can show that the convex hull of a set of points is exactly the set of all possible convex combinations of the points. We can therefore test whether a mixture can be obtained from the base mixtures by computing the convex hull of their representative points, and checking whether the point representing the mixture lies inside it.

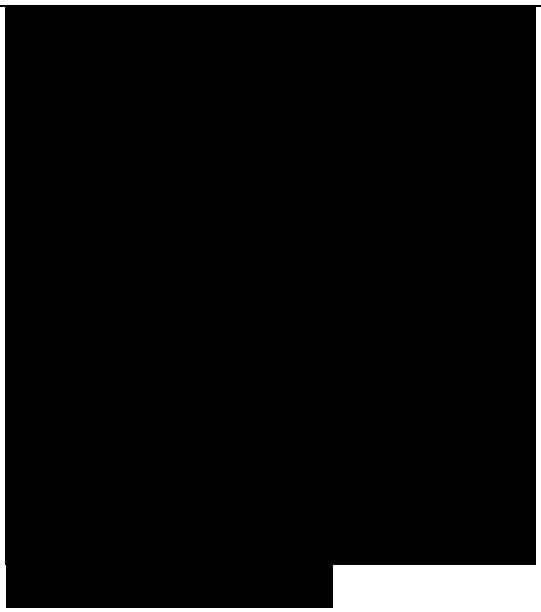
What if there are more than two interesting components in the mixtures? Well, what we have said above remains true; we just have to move to a space of higher dimension. More precisely, if we want to take d components into account we have to represent a mixture by a point in d-

dimensional space. The convex hull of the points representing the base mixtures, which is a convex polytope, represents the set of all possible mixtures.

Convex hulls—in particular convex hulls in 3-dimensional space—are used in various applications. For instance, they are used to speed up collision detection in computer animation. Suppose that we want to check whether two objects P1 and P2 intersect. If the answer to this question is negative most of the time, then the following strategy pays off. Approximate the objects by simpler objects P1 and P2 that contain the originals. If we want to check whether P1 and P2 intersect, we first check whether P1 and P2 intersect; only if this is the case do we need to perform the—supposedly more costly—test on the original objects.

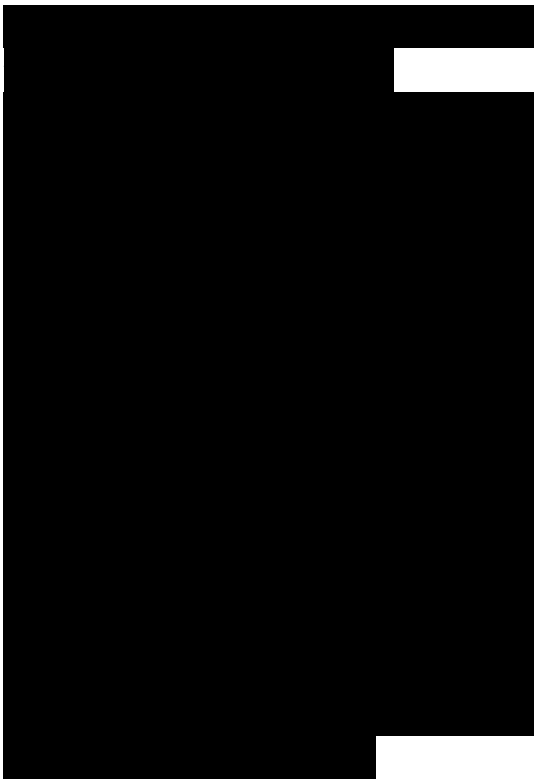
There is a trade-off in the choice of the approximating objects. On the one hand, we want them to be simple so that intersection tests are cheap. On the other hand, simple approximations most likely do not approximate the original objects very well, so there is a bigger chance we

have to test the originals. Bounding spheres are on one side of the spectrum: intersection tests for spheres are quite simple, but for many objects spheres do not provide a good approximation. Convex hulls are more on the other side of the spectrum: intersection tests for convex hulls are more complicated than for spheres—but still simpler than for non-convex objects—but convex hulls can approximate most objects a lot better.



11.1 The Complexity of Convex Hulls in 3-Space

In Chapter 1 we have seen that the convex hull of a set P of n points in the plane is a convex polygon whose vertices are points in P. Hence, the convex hull has at most n vertices. In 3-dimensional space a similar statement is true: the convex hull of a set P of n points is a convex polytope whose vertices are points in P and, hence, it has at most n vertices. In the planar case the bound on the number of vertices immediately implies that the complexity of the convex hull is linear, since the number of edges of a planar polygon is equal to the number of vertices. In 3-



space this is no longer true; the number of edges of a polytope can be higher than the number of vertices. But fortunately the difference cannot be too large, as follows from the following theorem on the number of edges and facets of convex polytopes. (Formally, a facet of a convex polytope is defined to be a maximal subset of coplanar points on its boundary. A facet of a convex polytope is necessarily a convex polygon. An edge of a convex polytope is an edge of one of its facets.)

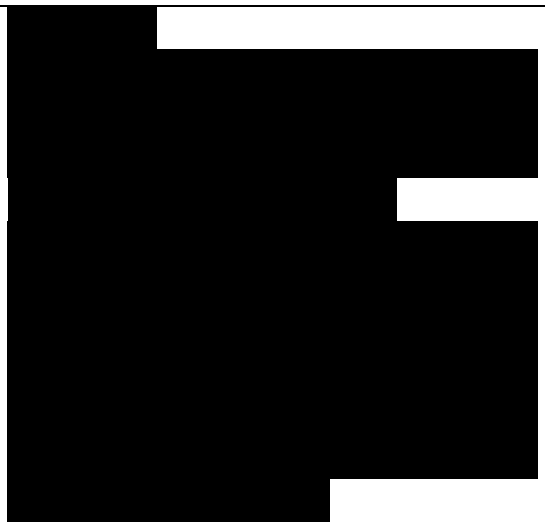
Theorem 11.1 Let P be a convex polytope with n vertices. The number of edges of P is at most 3n — 6, and the number of facets of P is at most 2n — 4.

Proof. Recall that Euler's formula states for a connected planar graph with n nodes, ne arcs, and nf faces the following relation holds:

n — ne + nf = 2.

Since we can interpret the boundary of a convex polytope as a planar graph—see Figure 11.1—the same relation holds for the numbers of vertices, edges, and facets in a convex polytope. (In fact, Euler's formula was originally stated in

Figure 11.1
A cube interpreted as a planar graph: note that one facet maps to the unbounded face of the graph



terms of polytopes, not in terms of planar graphs.) Every face of the graph corresponding to P has at least three arcs, and every arc is incident to two faces, so we have $2ne > 3nf$. Plugging this into Euler's formula we get

$$n + nf - 2 > 3nf/2,$$

so $nf < 2n - 4$. Applying Euler's formula once more, we see that $ne < 3n - 6$.

For the special case that every facet is a triangle—the case of a simplicial polytope—the bounds on the number of edges and facets of an n-vertex polytope are exact, because then $2ne = 3nf$. ED

Theorem 11.1 also holds for non-convex polytopes whose so-called genus is zero, that is, polytopes without holes or tunnels; for polytopes of larger genus similar bounds hold. Since this chapter deals with convex hulls, however, we refrain from defining what a (non-convex) polytope exactly is, which we would need to do to prove the theorem in the non-convex case.

If we combine Theorem 11.1 with the earlier observation that the convex hull of a set of points in 3-space is a convex polytope whose vertices are points in P, we get the following result.

Corollary 11.2 The complexity of the convex hull of a set of n points in three dimensional space is O(n).

11.2 Computing Convex Hulls in 3-Space

Let P be a set of n points in 3-space. We will compute CH(P), the convex hull of P, using a randomized incremental algorithm, following the paradigm we have met before in Chapters 4, 6, and 9.

The incremental construction starts by choosing four points in P that do not lie in a common plane, so that their convex hull is a tetrahedron. This can be done as follows. Let p1 and p2 be two points in P. We walk through the set P until we find a point p3 that does not lie on the line through p1 and p2. We continue searching P until we find a point p4 that does not lie in the plane through p1, p2, and p3. (If we cannot find

four such points, then all points in P lie in a plane. In this case we can use the planar convex hull algorithm of Chapter 1 to compute the convex hull.)

Next we compute a random permutation p5,... , pn of the remaining points. We will consider the points one by one in this random order, maintaining the convex hull as we go. For an integer r > 1, let Pr := {p1,...,pr}. In a generic step of the algorithm, we have to add the point pr to the convex hull of Pr-1, that is, we have to transform CH(Pr-1) into CH(Pr). There are two cases.

■ If pr lies inside CH(Pr-1), or on its boundary, then CH(Pr) = CH(Pr-1), and there is nothing to be done.

Figure 11.2 The horizon of a polytope

Now suppose that pr lies outside CH(Pr-1). Imagine that you are standing at pr, and that you are looking at CH(Pr-1). You will be able to see some facets of CK(Pr—1)—the ones on the front side—but others will be invisible because they are on the back side. The visible

facets form a connected region on the surface of CH(Pr—1), called the visible region of pr on CH(Pr—1), which is enclosed by a closed curve consisting of edges of CH(Pr—1). We call this curve the horizon of pr on CK(Pr— 1). As you can see in Figure 11.2, the projection of the horizon is the boundary of the convex polygon obtained by projecting CK(Pr—1) onto a plane, with pr as the center of projection. What exactly does "visible" mean geometrically? Consider the plane hf containing a facet f of CK(Pr— 1). By convexity, CK(Pr— 1) is completely contained in one of the closed half-spaces defined by hf. The face f is visible from a point if that point lies in the open half-space on the other side of h f.

The horizon of pr plays a crucial role when we want to transform CK(Pr— 1) to CH(Pr): it forms the border between the part of the boundary that can be kept—the invisible facets—and the part of the boundary that must be replaced—the visible facets. The visible facets must be replaced by facets connecting pr to its horizon.

Before we go into more details, we should decide how we are going to represent the convex hull of points in space. As we observed before, the boundary of a 3-dimensional convex polytope can be interpreted as a planar graph. Therefore we store the convex hull in the form of a doubly-connected edge list, a data structure developed in Chapter 2 for storing planar subdivisions. The only difference is that vertices will now be 3-dimensional points. We will keep the convention that the half-edges are directed such that the ones bounding any face form a counterclockwise cycle when seen from the outside of the poly tope.

Figure 11.3
Adding a point to the convex hull

Back to the addition of pr to the convex hull. We have a doubly-connected edge list representing CK(Pr— 1), which we have to transform into a doubly- connected edge list for CK(Pr). Suppose that we knew all facets of CK(Pr— 1) visible from pr.

Then it would be easy to remove all the information stored for these facets from the doubly-connected edge list, compute the new facets connecting pr to the horizon, and store the information for the new facets in the doubly-connected edge list. All this will take linear time in the total complexity of the facets that disappear.

There is one subtlety we should take care of after the addition of the new facets: we have to check whether we have created any coplanar facets. This happens if pr lies in the plane of a face of CH(Pr-1). Such a face f is not visible from pr by our definition of visibility above. Hence, f will remain unchanged, and we will add triangles connecting pr to the edges of f that are part of the horizon. Those triangles are coplanar with f, and so they have to be merged with f into one facet.

In the discussion so far we have ignored the problem of finding the facets of CH(Pr-1) that are visible from pr. Of course this could be done by testing every facet. Since such a test takes constant time—we have to check to

which side of a given plane the point pr lies—we can find all visible facets in O(r) time. This would lead to an O(n2) algorithm. Next we show how to do better.

The trick is that we are going to work ahead: besides the convex hull of the current point set we shall maintain some additional information, which will make it easy to find the visible facets.
In particular, we maintain for each facet f of the current convex hull CH(Pr) a set Pconflict(f) c {pr+1, pr+2,...,pn} containing the points that can see f. Conversely, we store for every point pt, with t > r, the set Fconflict(pt) of facets of CH(Pr) visible from pt. We will say that a point p e Pconflict(f) is in conflict with the facet f, because p and f cannot peacefully live together in the convex hull—once we add a point p e Pconflict(f) to the convex hull, the facet f must go. We call Pconflict(f) and Fconflict(pt) conflict lists.

We maintain the conflicts in a

so-called conflict graph, which we denote by S. The conflict graph is a bipartite graph. It has one node set with a node for every point of P that has not been inserted yet, and one node set with a node for every facet of the current convex hull. There is an arc for every conflict between a point and a facet. In other words, there is an arc between a point pt e P and facet f of CH(Pr) if r < t and f is visible from pt. Using the conflict graph S, we can report the set Fconflict(pt) for a given point pt (or Pconflict(f) for a given facet f) in time linear in its size.

This means that when we insert pr into CH(Pr-1), all we have to do is to look up Fconflict(pr) in S to get the visible facets, which we can then replace by the new convex hull facets connecting pr to the horizon.

Initializing the conflict graph S for CH(P4) can be done in linear time: we simply walk through the list of points P and determine which of the four faces of CH(P4) they can see.

To update S after adding a point pr, we first discard the nodes and incident arcs for all the facets of CH(Pr-1) that disappear from the convex hull. These are the facets visible from pr, which are exactly the neighbors of pr in S, so this is easy. We also discard the node for pr. We then add nodes to S for the new facets we created, which connect pr to the horizon. The essential step is to find the conflict lists of these new facets. No other conflicts have to be updated: the conflict set Pconflict(f) of a facet f that is unaffected by the insertion of pr remains unchanged.

The facets created by the insertion of pr are all triangles, except for those that have been merged with existing coplanar facets. The conflict list of a facet of the latter type is trivial to find: it is the same as the conflict list of the existing facet, since the merging does not change the plane containing the facet. So let's look at one of the new triangles f incident to pr in CK(Pr). Suppose that a point pt can see f. Then pt can certainly see the edge e of f that is opposite pr. This edge e is a horizon edge of pr, and it was already present in

CK(Pr— 1). Since CK(Pr— 1) c CK(Pr), the edge e must have been visible from pt in CK(Pr— 1) as well. That can only be the case if one of the two facets incident to e in CK(Pr— 1) is visible from pt. This implies that the conflict list of f can be found by testing the points in the conflict lists of the two facets f1 and f2 that were incident to the horizon edge e in CK(Pr— 1).

We stated earlier that we store the convex hull as a doubly-connected edge list, so changing the convex hull means changing the information in the doubly-connected edge list. To keep the code short, however, we have omitted all explicit references to the doubly-connected edge list in the pseudocode below, which summarizes the convex hull algorithm.

Algorithm CONVEXHULL(P)

Input. A set P of n points in three-space.

Output. The convex hull CK(P) of P.

1. Find four points p1, p2, p3, p4 in P that form a tetrahedron.
2. e ^ CK({p1, p2, p3, p4»
3. Compute a random

permutation p5, p6,..., pn of the remaining points.
4. Initialize the conflict graph G with all visible pairs (pt, f), where f is a facet of e and t > 4.

5. for r^- 5 to n
6. do (* Insert pr into C: *)
7. if Fconflict(pr) is not empty (* that is, pr lies outside e *)

8. then Delete all facets in Fconflict(pr) from e.

9. Walk along the boundary of the visible region of pr (which consists exactly of the facets in Fconflict(pr)) and create a list L of horizon edges in order.
10. for all e e L

11. do Connect e to pr by creating a triangular facet f.

12. if f is coplanar with its neighbor facet f' along e
13. then Merge f and f' into one facet, whose conflict list is the same as that of f.

14. else (* Determine conflicts for f: *)
15. Create a node for f in G.
16. Let f1 and f2 be the

facets incident to e in the old convex hull.

17. P(e) ^ Pconflict(f1) u Pconflict(f2)

18. for all points p e P(e)
19. do If f is visible from p, add (p, f) to G.

20. Delete the node corresponding to pr and the nodes corre-sponding to the facets in Fconflict(pr) from G, together with their incident arcs.

11.3* The Analysis
As usual when we analyse a randomized incremental algorithm, we first try to bound the expected structural change. For the convex hull algorithm this means we want to bound the total number of facets created by the algorithm.

Lemma 11.3 The expected number of facets created by ConvexHull is at most 6n _ 20.
Proof. The algorithm starts with a tetrahedron, which has four facets. In every stage r of the algorithm where pr lies outside CH(Pr-1), new triangular facets connecting pr to its horizon on CH(Pr-1)

are created. What is the expected number of new facets?

As in previous occasions where we analyzed randomized algorithms, we use backwards analysis. We look at CH(Pr) and imagine remov¬ing vertex pr; the number of facets that disappear due to the removal of pr from CH(Pr) is the same as the number of facets that were created due to the insertion of pr into CH(Pr-1). The disappearing facets are exactly the ones incident to pr, and their number equals the number of edges incident to pr in CH(Pr). We call this number the degree of pr in CH(Pr), and we denote it by deg(pr, CH(Pr)). We now want to bound the expected value of deg(pr, CH(Pr)).

By Theorem 11.1a convex polytope with r vertices has at most 3r _ 6 edges. This means that the sum of the degrees of the vertices of CH(Pr), which is a convex polytope with r or less vertices, is at most 6r _ 12. Hence, the average degree is bounded by 6 _ 12/r. Since we treat the vertices in random order, it seems that the expected degree of pr is bounded by 6 _ 12/r. We have

to be a little bit careful, though: the first four points are already fixed when we generate the random permutation, so pr is a random element of {p5,..pr}, not of Pr. Because p1,..., p4 have total degree at least 12, the expected value of deg(pr, CH(Pr)) is bounded as follows:

The expected number of facets created by CONVEXHULL is the number of facets we start with (four) plus the expected total number of facets created during the additions of p5,... , pn to the hull. Hence, the expected number of created facets is

$4 + n\,E[\deg(Pr, CH(Pr))] < 4 + 6(n \_ 4) = 6n \_ 20.$

r=5

Now that we have bounded the total amount of structural change we can bound the expected running time of the algorithm.

Lemma 11.4 Algorithm ConvexHull computes the convex hull of a set P of n points in R3 in O(nlog n) expected time, where the expectation is with respect to the random permutation used by the algorithm.

Proof. The steps before the main loop can certainly be

done in O(nlogn) time. Stage r of the algorithm takes constant time if Fconflict(pr) is empty, which is when pr lies inside, or on the boundary of, the current convex hull.
If that is not the case, most of stage r takes O(card(Fconflict(pr))) time, where card() denotes the cardinality of a set. The exceptions to this are the lines 17-19 and line 20. We shall bound the time spent in these lines later; first, we bound card(Fconflict(pr)). Note that card(Fconflict(pr)) is the number of facets deleted due to the addition of the point pr. Clearly, a facet can only be deleted if it has been created before, and it is deleted at most once. Since the expected number of facets created by the algorithm is O(n) by Lemma 11.3, this implies that the total number of deletions is O(n) as well, so
Now for lines 17-19 and line 20. Line 20 takes time linear in the number of nodes and arcs that are deleted from G . Again, a node or arc is deleted at most once, and we can charge the cost of this deletion to the stage where we created it. It remains to look at lines 17-19. In stage r, these lines are executed for

all horizon edges, that is, all edges in L.

For one edge e e L, they take O(card(P(e))) time. Hence, the total time spent in these lines in stage r is O(£eeL card(P(e))). To bound the total expected running time, we therefore have to bound the expected value of

where the summation is over all horizon edges that appear at any stage of the algorithm. We will prove below that this is O(nlogn), which implies that the total running time is O(nlog n). 0

We use the framework of configuration spaces from Chapter 9 to supply the missing bound. The universe X is the set of P, and the configurations A correspond to convex hull edges. However, for technical reasons—in particular, to be able to deal correctly with degenerate cases—we attach a half-edge to both sides of the edge. To be more precise, a flap A is defined as an ordered four-tuple of points (p, q, s, t) that do not all lie in a plane. The defining set D(A) is simply the set {p, q,

s, t}. The killing set K(A) is more difficult to visualize. Denote the line through p and q by I. Given a point X, let h(l,x) denote the half-plane bounded by I that contains x. Given two points x, y, let p (x, y) be the half-line starting in x and passing through y. A point x e X is in K(A) if and only if it lies in one of the following regions:

- outside the closed convex 3-dimensional wedge defined by h(£, s) and h(£, t),
- inside h(£, s) but outside the closed 2-dimensional wedge defined by p (p, q) and p (p, s),
- inside h(t, t) but outside the closed 2-dimensional wedge defined by p (q, t) and p (q, p),
- inside the line I but outside the segment pq,
- inside the half-line p (p, s) but outside the segment ps,
- inside the half-line p (q, t) but outside the segment qt.

For every subset S c P, we define the set T(S) of active configurations—this is what we want to compute—as prescribed in Chapter 9: A e

T(S) if and only if D(A) c S and K(A) n S = 0.

Lemma 11.5 A flap A = (p, q, s, t) is in T(S) if and only if pq, ps, and qt are edges of the convex hull CH(S), there is a facet f1 incident to pq and ps, and a different facet f2 incident to pq and qt. Furthermore, if one of the facets f1 or f2 is visible from a point x e P then x e K(A).

We leave the proof—which involves looking precisely at the cases when points are collinear or coplanar, but which is otherwise not difficult—to the reader.

As you may have guessed, the flaps take over the role of the horizon edges.

Lemma 11.6 The expected value of £ecard(P(e)), where the summation is over all horizon edges that appear at some stage of the algorithm, is O(nlogn).

Proof. Consider an edge e of the horizon of pr on CH(Pr-1). Let A = (p, q, s, t) be one of the two flaps with pq = e. By Lemma 11.5, A e T(Pr-1), and the points in P \ Pr that can see one of the facets incident to e are all in K(A),

so P(e) c K(A). By Theorem 9.15, it follows that the expected value of

where the summation is over all flaps A appearing in at least one T(Pr), is bounded by

The cardinality of T(Pr) is twice the number of edges of CH(Pr). Therefore it is at most 6r _ 12, so we get the bound
This finishes the last piece of the analysis of the convex hull algorithm. We get the following result:

Theorem 11.7 The convex hull of a set of n points in R3 can be computed in O(nlog n) randomized expected time.

11.4* Convex Hulls and Half-Space Intersection
In Chapter 8 we have met the concept of duality. The strenth of duality lies in that it allows us to look at a problem from a new perspective, which can lead to more insight in what is really going on. Recall that we denote the line that is the dual of a point p by p*, and the point that is the dual of a line I by I*. The duality transform is incidence and order preserving: p e I if

and only if I* e p*, and p lies above I if and only if I* lies above p*.

Let's have a closer look at what convex hulls correspond to in dual space. We will do this for the planar case. Let P be a set of points in the plane. For technical reasons we focus on its upper convex hull, denoted UH(P), which consists of the convex hull edges that have P below their supporting line—see the left side of Figure 11.4. The upper convex hull is a polygonal chain that connects the leftmost point in P to the rightmost one. (We assume for simplicity that no two points have the same x-coordinate.)

Figure 11.4
Upper hulls correspond to lower envelopes

When does a point p e P appear as a vertex of the upper convex hull? That is the case if and only if there is a non-vertical line I through p such that all other points of P lie below I. In the dual plane this statement translates to the following condition: there is a point I* on the line p* e P* such that I* lies below all other lines of P*. If we look at the arrangement A(P*), this

means that p* contributes an edge to the unique bottom cell of the arrangement. This cell is the intersection of the half-planes bounded by a line in P* and lying below that line. The boundary of the bottom cell is an x-monotone chain. We can define this chain as the minimum of the linear functions whose graphs are the lines in P*.

For this reason, the boundary of the bottom cell in an arrangement is often called the lower envelope of the set of lines. We denote the lower envelope of P* by LE(P*)—see the right hand side of Figure 11.4.

The points in P that appear on UH(P) do so in order of increasing x- coordinate. The lines of P* appear on the boundary of the bottom cell in order of decreasing slope. Since the slope of the line p* is equal to the x-coordinate of p, it follows that the left-to-right list of points on UH(P) corresponds exactly to the right-to-left list of edges of LE(P*). So the upper convex hull of a set of points is essentially the same as the lower envelope of a set of lines.

Let's do one final check. Two

points p and q in P form an upper convex hull edge if and only if all other points in P lie below the line i through p and q. In the dual plane, this means that all lines r*, with r e P \{p, q}, lie above the intersection point i* of p* and q*. This is exactly the condition under which p*n q* is a vertex of LE(P*).

What about the lower convex hull of P and the upper envelope of P*? (We leave the precise definitions to the reader.) By symmetry, these concepts are dual to each other as well.

We now know that the intersection of lower half-planes—half-planes bounded from above by a non-vertical line—can be computed by computing an upper convex hull, and that the intersection of upper half-planes can be computed by computing a lower convex hull. But what if we want to compute the intersection of an arbitrary set H of half-planes? Of course, we can split the set H into a set H+ of upper half-planes and a set H- of lower half-planes, compute u H+ by computing the lower convex hull of H+* and uH- by computing the upper convex hull of H-*, and then compute

nH by intersecting uH+ and uH-.
But is this really necessary? If lower envelopes correspond to upper convex hulls, and upper envelopes correspond to lower convex hulls, shouldn't then the intersection of arbitrary half-planes correspond to full convex hulls? In a sense, this is true. The problem is that our duality transformation cannot handle vertical lines, and lines that are close to vertical but have opposite slope are mapped to very different points. This explains why the dual of the convex hull consists of two parts that lie rather far apart.

It is possible to define a different duality transformation that allows vertical lines. However, to apply this duality to a given set of half-planes, we need a point in the intersection of the half-planes. But that was to be expected. As long as we do not want to leave the Euclidean plane, there cannot be any general duality that turns the intersection of a set of half-planes into a convex hull, because the intersection of half-planes can have one special property: it can be

empty. What could that possibly correspond to in the dual? The convex hull of a set of points in Euclidean space is always well defined: there is no such thing as "emptiness." (This problem is nicely solved if one works in oriented projective space, but this concept is beyond the scope of this book.) Only once you know that the intersection is not empty, and a point in the interior is known, can you define a duality that relates the intersection with a convex hull.

We leave it at this for now. The important thing is that—although there are technical complications—convex hulls and intersections of half-planes (or half¬spaces in three dimensions) are essentially dual concepts.

Hence, an algorithm to compute the intersection of half-planes in the plane (or half-spaces in three dimensions) can be given by dualizing a convex-hull algorithm.

11.5* Voronoi Diagrams Revisited

In Chapter 7 we introduced the Voronoi diagram of a set of points in the plane. It may come as a surprise that there is a close relationship between planar Voronoi diagrams and the intersection of upper half-spaces in 3-dimensional space. By the result on duality of the previous section, this implies a close relation between planar Voronoi diagrams and lower convex hulls in 3-space.

This has to do with an amazing property of the unit paraboloid in 3-space. Let U := (z = x2 + y2) denote the unit paraboloid, and let p := (px,py, 0) be a point in the plane z = 0. Consider the vertical line through p. It intersects U in the point p' := (px,py,pi + pi). Let h(p) be the non-vertical plane z = 2pxx + 2pyy _ (p2x + p2). Notice that h(p) contains the point p'. Now consider any other point q := (qx, qy, 0) in the plane z = 0. The vertical line through q intersects U in the point q' := (qx, qy, ql + qi), and it intersects h(p) in q(p) := (qx, qy, 2pxqx + 2pyqy _ (pi + p2)).

The vertical distance between q' and q(p) is

ql + q2 _ 2pxqx _ 2pyqy + pi + pi = (qx _ px)2 + (qy _ pyf = dist(p, q)2.

Hence, the plane h(p) encodes—together with the unit paraboloid—the distance between p and any other point in the plane z = 0. (Since dist(p, q)2 > 0 for any point q, and p' e h(p), this also implies that h(p) is the tangent plane to U at p'.)

The fact that the plane h(p) encodes the distance of other points to p leads to a correspondence between Voronoi diagrams and upper envelopes, as explained next. Let P be a planar point set, which we imagine to lie in the plane z = 0 of 3-dimensional space. Consider the set H := {h(p) I p e P} of planes, and let UE(H) be the upper envelope of the planes in H. We claim that the projection of UE(H) on the plane z = 0 is the Voronoi diagram of P. Figure 11.5 illustrates this one dimension lower: the Voronoi diagram of the points pi on the line y = 0 is the projection of the upper envelope of the lines h(pi).

Theorem 11.8 Let P be a set

of points in 3-dimensional space, all lying in the plane z = 0. Let H be the set of planes h(p), for p e P, deũned as above. Then the projection of UE(H) on the plane z = 0 is the Voronoi diagram of P.

Proof. To prove the theorem, we will show that the Voronoi cell of a point p e P is exactly the projection of the facet of UE(H) that lies on the plane h(p). Let q be a point in the plane z = 0 lying in the Voronoi cell of p. Hence, we have dist(q, p) < dist(q, r) for all r e P with r = p. We have to prove that the vertical line through q intersects UE(H) at a point lying on h(p). Recall that for a point r e P, the plane h(r) is intersected by the vertical line through q at the point q(r) := (qx, qy, qx + q2 _ dist(q, r)2). Of all points in P, the point p has the smallest distance to q, so q(p) is the highest intersection point. Hence, the vertical line through q intersects UE(H) at a point lying on h(p), as claimed.

This theorem implies that we can compute a Voronoi diagram in the plane by computing the upper

envelope of a set of planes in 3-space. By Exercise 11.10 (see also the previous section), the upper envelope of a set of planes in 3-space is in one-to-one correspondence to the lower convex hull of the points H*, so we can immediately use our algorithm ConvexHull.

Figure 11.5
The correspondence between Voronoi diagrams and upper envelopes

Not surprisingly, the lower convex hull of H* has a geometric meaning as well: its projection on the plane z = 0 is the Delaunay graph of P.

11.6 Notes and Comments

The early convex hull algorithms worked only for points in the plane—see the notes and comments of Chapter 1 for a discussion of these algorithms. Computing convex hulls in 3-dimensional space turns out to be considerably more difficult. One of the first algorithms was the "gift wrapping" algorithm due to Chand and Kapur [84]. It finds facet after facet by "rotating" a plane

over known edges of the hull until the first point is found. The running time is O(nf) for a convex hull with f facets, which is O(n2) in the worst case. The first algorithm to achieve O(nlog n) running time was a divide-and-conquer algorithm by Preparata and Hong [322, 323]. Early incremental algorithms run in time O(n2) [223, 344]. The randomized version presented here is due to Clarkson and Shor [133]. The version we presented needs O(nlogn) space; the original paper gives a simple improvement to linear space. The idea of a conflict graph, used here for the first time in this book, also comes from the paper of Clarkson and Shor. Our analysis, however, is due to Mulmuley [290].

In this chapter we have concentrated on 3-dimensional space, where convex hulls have linear complexity. The so-called Upper Bound Theorem states that the worst-case combinatorial complexity of the convex hull of n points in d- dimensional space—phrased in dual space: the intersection of n half-spaces—is 0(nLd/2J). (We proved this result for the case

d = 3, using Euler's relation.) The algorithm described in this chapter generalizes to higher dimensions, and is optimal in the worst case: its expected running time is 0(n^d/2J). Interestingly, the best known deterministic convex hull algorithm for odd-dimensional spaces is based on a (quite complicated) derandomization of this algorithm [97]. Since the convex hull in dimensions greater than three can have non-linear complexity, output-sensitive algorithms may be useful. The best known output-sensitive algorithm for computing convex hulls in Rd is due to Chan [82]. Its running time is O(nlogk + (nk)1-1/(Ld/2J+1) logO(1) n), where k denotes the complexity of the convex hull. A good overview of the many results on convex-hull computations is given in the survey by Seidel [347]. Readers who want to know more about the mathematical aspects of polytopes in higher dimensions can consult Grunbaum's book [194], which is a classical reference for polytope theory, or Ziegler's book [399], which treats the combinatorial aspects.	[redacted]	

In Section 11.5 we have seen that the Voronoi diagram of a planar point set is the projection of the upper envelope of a certain set of planes in 3-dimensional space. A similar statement is true in higher dimensions: the Voronoi diagram of a set of points in Rd is the projection of the upper envelope of a certain set of hyperplanes in Rd+1. Not all sets of (hyper)planes define an upper envelope whose projection is the Voronoi diagram of some point set. Interestingly, any upper envelope does project onto a so-called power diagram, a generalization of the Voronoi diagram where the sites are spheres rather than points [25].